Revento

Mohamad Hatem Zrek

Table of Contents

Figures and Table

1. Introduction

Revento is an event management and ticket booking platform designed to provide users with seamless experience in discovering, booking, and managing events. The platform features personalized event recommendations, user preferences, and a secure authentication system. Built with Next.js (TypeScript) for the frontend**,** PHP (MySQL) for the backend, and AI-powered recommendations**.**

---

2. Problem Statement

The event ticketing platform requires a recommendation system that:

- Suggests the top 3 events for each user based on their interests and past bookings.
- Uses optimized data structures and ranking algorithms to enhance recommendation quality.
- Scales efficiently for millions of users and events.
- Handles the cold-start problem for new users and new event types.

---

3. Approach & Methodology

3.1 Data Model

The recommendation system relies on structured data related to users, events, bookings, and preferences.

- **Users:** have unique identifiers and store authentication details, including their roles (admin or regular user).
- **Events:** contain information such as title, category, subcategory, location, and pricing. Categories and subcategories play a crucial role in filtering recommendations.
- **Bookings:** track which events users have attended or booked, helping to refine future recommendations.
- **User Preferences:** store each user's interests, including preferred event categories, subcategories, and city, enabling personalized recommendations.

The relationships between these data points help generate personalized event suggestions based on past user interactions and explicit preferences.

Figure 1 Database Schema

3.2 Recommendation Logic

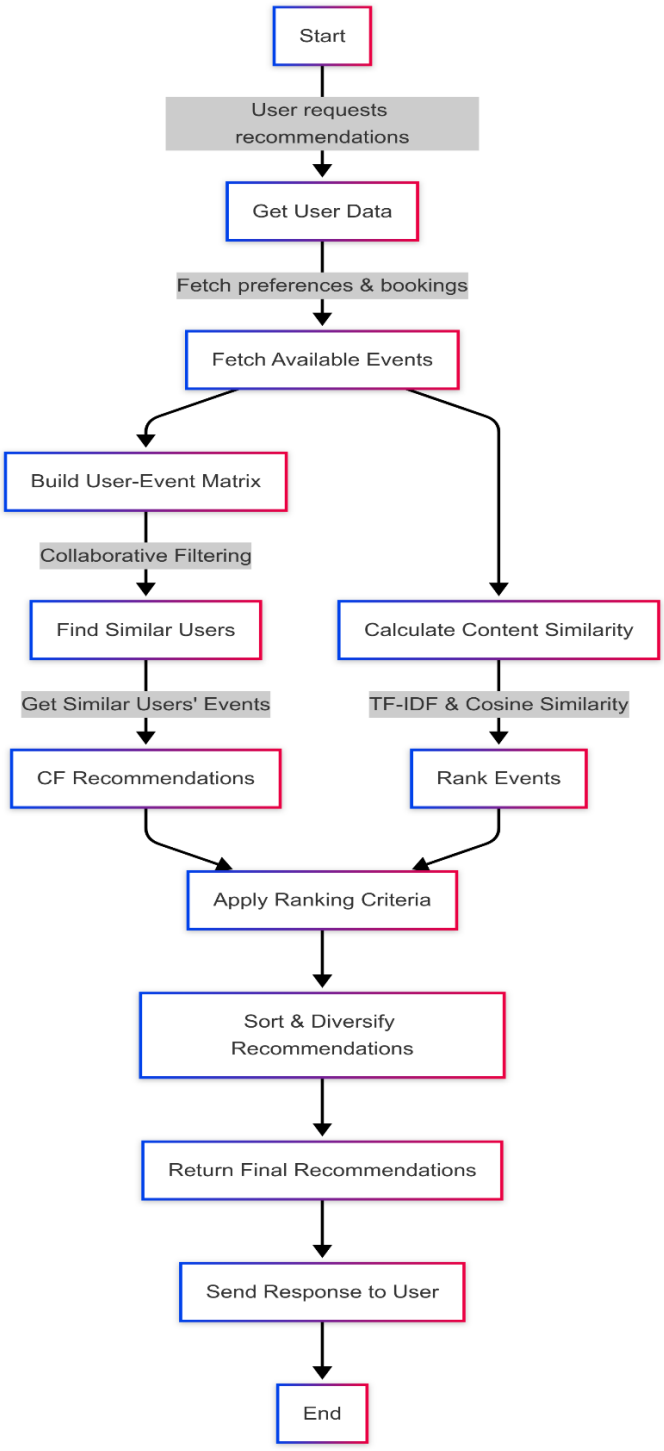

Figure 2 Recommendation System Flowchart

The event recommendation system uses a hybrid AI approach that combines collaborative filtering, content-based filtering (TF-IDF), and a personalized ranking algorithm to provide accurate and personalized event suggestions.

Step 1: Retrieve User Preferences & Booking History

- Fetch the user's preferences (category, subcategory, city) from the database.
- Retrieve the last 5 booked events to understand user interests.

Step 2: Collaborative Filtering (User-Based Similarity)

- Build a User-Event Matrix**:**
  - Each row represents a user, and each column represents an event.
  - A 1 in the matrix means the user booked that event.
- Compute Similarity Between Users**:**
  - Use cosine similarity to find users with similar booking patterns.
  - Identify top 5 most similar users.
  - Recommend top 5 events booked by those users.

Step 3: Content-Based Filtering (TF-IDF)

- Convert event categories into numerical vectors using TF-IDF (Term Frequency-Inverse Document Frequency)**.**
- Compute similarity scores between user preferences and available events.
- Events matching preferred categories and subcategories receive a higher-ranking score**.**

Step 4: Personalized Ranking System

Each event is assigned a ranking score based on multiple factors:

Table 1 Ranking System

Factor	Score Weight
Preference Match (Category/Subcategory)	+40 (Exact), +20 (Partial)
Past Booking Similarity (Same category)	+30
Location Match	+30
New User Boost	+25
Event Similarity Score (TF-IDF)	+0 to +40
Event Popularity (Bookings Count)	+5 to +30
Event Date Boost (Recency)	+15 (7 days), +10 (30 days)
Collaborative Filtering Boost	+70
Random Factor (Tiebreaker)	+0 to +10

Step 5: Final Selection & Diversity Handling

- Sort events by highest ranking score.
- Remove already booked events to avoid redundancy.
- Enforce category diversity (max 2 events per category) to maintain variety.
- Select the top 3 events for the final recommendation.

---

4. Technical Implementation

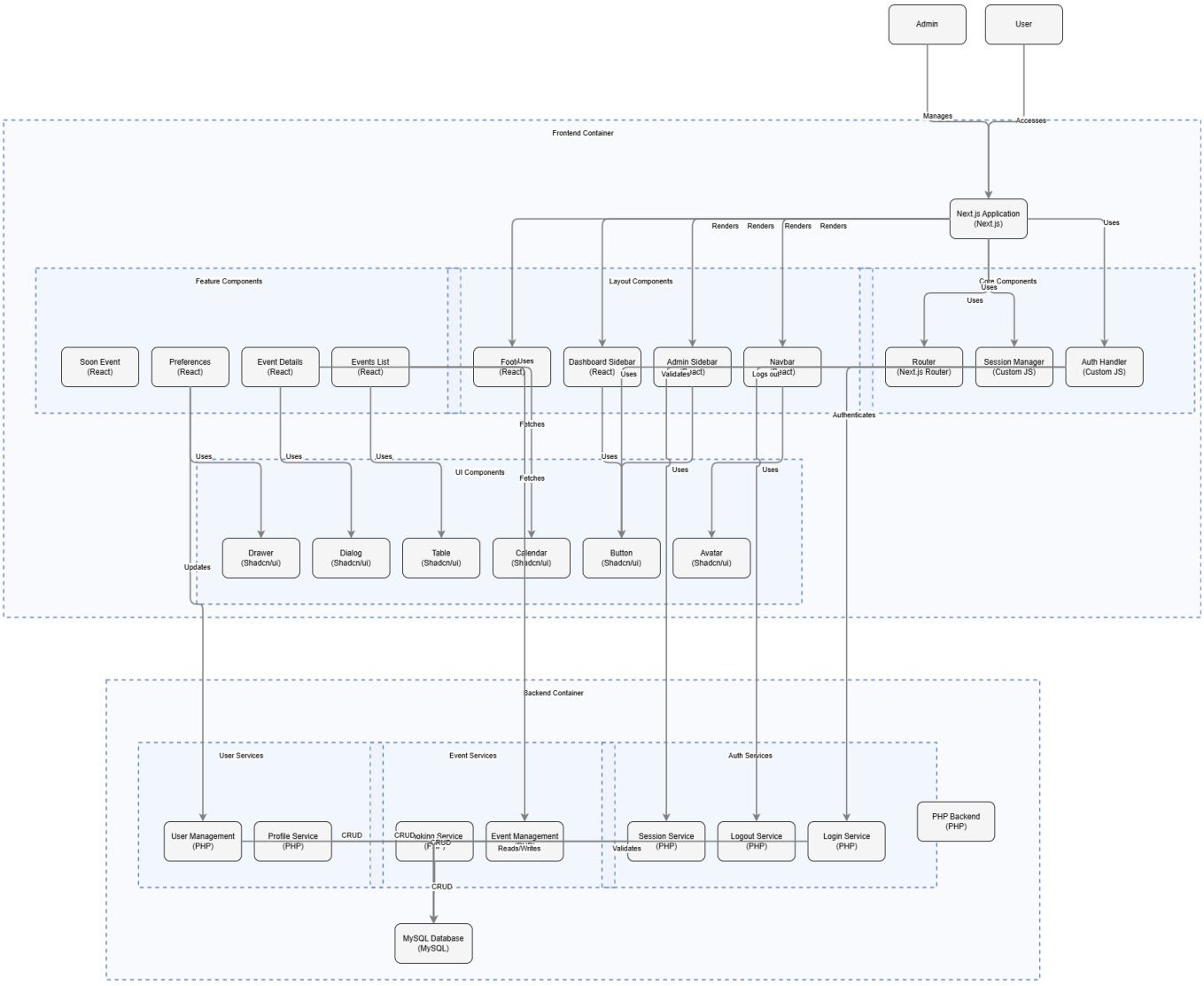

Figure 3 System Architecture Diagram

4.1 Data Structures

- User-Event Matrix (Collaborative Filtering) → Pandas DataFrame (Sparse Matrix) to store user-event interactions for similarity calculations.
- TF-IDF Vectorization (Content-Based Filtering) → Scikit-learns TfidfVectorizer to analyze event category similarities.
- Event Ranking Dictionary → Python Dictionary {event_id: score} to store final ranking scores.

4.2 Algorithm Pseudocode

1-Retrieve Data

```
preferences, past_bookings = get_user_data(user_uuid)
events = get_upcoming_events()
```

2-Collaborative Filtering (User-Based Recommendations)

```
user_event_matrix = build_user_event_matrix()
similar_users = get_similar_users(user_uuid, user_event_matrix)
collaborative_recommendations = get_top_events_from_similar_users(similar_users)
```

3-Content-Based Filtering (TF-IDF)

```
similarity_matrix = compute_tfidf_similarity(events)
```

4-Compute Event Ranking Score

- Preference Match: +40 (Exact) / +20 (Partial)
- Past Booking Similarity: +30
- Location Match: +30
- New User Boost: +25
- Category Similarity (TF-IDF Score): Max +40
- Event Popularity (Booking Count): Max +30
- Event Recency Boost: +15 (7 days) / +10 (30 days)
- Collaborative Filtering Boost: +70
- Random Factor (Tie-Breaker): +0 to +10

```
FOR event IN events:
   score = calculate_event_score(event, preferences, past_bookings, similarity_matrix,
collaborative_recommendations)
   event_scores[event["event_id"]] = score
```

5-Select Top 3 Events

Sort by Ranking Score and Remove Already Booked Events

```
sorted_events = sort_by_score(event_scores)
recommended_events = filter_booked_and_diversify(sorted_events, past_bookings)
RETURN recommended_events[:3]
```

---

5. Scalability Considerations

To ensure the recommendation system performs efficiently at scale, several optimizations are implemented. Database performance is enhanced through indexing on frequently queried columns, table partitioning by date or location, and caching mechanisms like Redis to reduce redundant queries. Data processing is optimized with batch computation, asynchronous job queues (Celery with Flask), and incremental updates to avoid unnecessary recalculations.

For machine learning computations, sparse matrix storage and dimensionality reduction techniques (SVD, PCA) are used to minimize memory usage, while distributed processing with Apache Spark ensures scalability for large datasets. API performance is improved through rate limiting, pagination, and load balancing to handle high traffic efficiently.

Cold-start issues are mitigated by recommending trending events for new users and leveraging content-based filtering for new event types. A hybrid approach combining collaborative filtering and popularity-based methods ensures relevant recommendations even in sparse datasets. These strategies maintain system responsiveness, accuracy, and efficiency as the platform scales.

---

6. Performance Optimization

Database Optimization

- **Indexing & Partitioning**: Speeds up queries by indexing frequently used columns and partitioning large tables.
- **Caching with Redis**: Reduces database load by storing frequently accessed recommendations.
- **Optimized Queries**: Uses prepared statements and minimizes complex joins to improve query execution time.

Efficient Recommendation Computation

- **Precomputed Similarity Scores**: Reduces real-time calculations by storing user-event similarity scores in advance.
- **Asynchronous Processing**: Background jobs (Celery with Flask) handle recommendations without blocking API requests.

- **Incremental Updates**: Recalculates only necessary data instead of reprocessing the entire dataset.

Machine Learning Optimization

- **Sparse Matrices for CF**: Reduces memory overhead in collaborative filtering.
- **Dimensionality Reduction (SVD, PCA)**; Improves efficiency by lowering computational complexity.
- **Parallel Processing**; Distributes workloads across multiple processors for faster execution.

API Performance Enhancements

- **Rate Limiting & Throttling**: Prevents excessive API calls and maintains system stability.
- **Response Caching**: Stores previous recommendations to reduce redundant computations.
- **Pagination & Lazy Loading**: Ensures efficient data delivery without overloading the system.

---

7. Future Plan

To further enhance the recommendation system and user experience, several improvements can be implemented.

One key enhancement is integrating cookies into the website to track user interactions, such as event views, clicks, and time spent on event pages. This data can be used to refine recommendations based on real-time user behavior, improving personalization beyond past bookings and preferences.

Additionally, AI-driven recommendation models can be introduced, such as deep learning-based collaborative filtering, which adapts to changing user interests over time. User feedback mechanisms, like allowing users to rate or mark events as "Interested," can further refine recommendations.

For performance improvements, edge caching and CDN integration can be utilized to deliver faster API responses. Expanding the system to support real-time recommendations and social features, such as friend-based event suggestions, can enhance engagement.

By incorporating these advancements, the platform can evolve into a more intelligent, responsive, and user-centric event discovery system

Mohamad Hatem Zrek