Data Pipelines Documentation

This document details the main data pipelines in the SmartTrip project, describing how data flows through the system from user interactions to database storage and analytics.

Table of Contents

Recommendation Pipeline

The recommendation pipeline is the core data flow that transforms user preferences into personalized trip recommendations.

Flow Overview

```
User Clicks Search Button (Frontend)
    ↓
Form State Collection (React State)
    ↓
Event Tracking (trackSearchSubmit)
    ↓
URL Parameter Encoding
    ↓
Navigation to /search/results
    ↓
Results Page Reads URL Params
    ↓
POST /api/v2/recommendations
    ↓
Recommendation Service (recommendation.py)
    ↓
Database Query (TripOccurrence + TripTemplate)
    ↓
Scoring Algorithm
    ↓
Results Formatting
    ↓
Response to Frontend
    ↓
Results Display & Event Tracking
    ↓
Logging (RecommendationLogger)
```

Detailed Steps

1. Frontend: User Interaction & Form Submission

Location: `frontend/src/app/search/page.tsx`

a. Form State Collection When user clicks the "מצא את הטיול שלי" (Find My Trip) button, the `handleSearch()` function collects all form state:

- **Geographic filters**:
  - `selectedLocations` array (countries and continents)
  - Extracted into `countriesIds` (comma-separated country IDs)
  - Extracted into `continents` (comma-separated continent names)
- **Trip preferences**:
  - `selectedType` → `preferred_type_id` (single trip type ID)
  - `selectedThemes` → `preferred_theme_ids` (array of up to 3 theme tag IDs)
- **Constraints**:
  - `minDuration`, `maxDuration` → `min_duration`, `max_duration`
  - `maxBudget` → `budget`
  - `difficulty` → `difficulty` (1-3, or null)
- **Date filters**:
  - `selectedYear` → `year` ('all' or specific year)
  - `selectedMonth` → `month` ('all' or 1-12)

b. Event Tracking Before navigation, the frontend tracks the search submission:

- Calls `trackSearchSubmit(preferences, searchType)` from `tracking.service.ts`
- Classifies search type: `'exploration'` (< 2 filters) or `'focused_search'` (≥ 2 filters)
- Event is queued in `eventQueue` for batch sending
- Calls `flushPendingEvents()` to ensure events are sent before navigation

c. URL Parameter Encoding Preferences are encoded into URL query parameters:

```
const params = new URLSearchParams();
if (countriesIds) params.set('countries', countriesIds);
if (continents) params.set('continents', continents);
if (selectedType) params.set('type', selectedType.toString());
// ... etc
router.push(`/search/results?${params.toString()}`);
```

d. Navigation Next.js router navigates to `/search/results` with query parameters, preserving state in URL for:

- Back button support
- Shareable search URLs
- Browser history

2. Frontend: Results Page Initialization

Location: `frontend/src/app/search/results/page.tsx`

a. URL Parameter Parsing On mount, `useEffect` reads search parameters from URL:

```
const countries = searchParams.get('countries')?.split(',').map(Number) || [];
const continents = searchParams.get('continents')?.split(',') || [];
const type = searchParams.get('type');
// ... etc
```

b. Preferences Object Construction Rebuilds preferences object from URL params:

```
const preferences = {
  selected_countries: countries,
  selected_continents: continents,
  preferred_type_id: type ? Number(type) : undefined,
  preferred_theme_ids: themes,
  min_duration: minDuration,
  max_duration: maxDuration,
  budget: budget,
  difficulty: difficulty,
  year: year || 'all',
  month: month || 'all',
};
```

c. API Request Makes POST request to backend via `getRecommendations()` from `frontend/src/api/v2.ts` :

- Function: `getRecommendations(preferences)`
- Endpoint: `POST /api/v2/recommendations`
- Uses `apiFetch()` wrapper from `client.ts` :
  - Adds authentication headers (if user logged in)
  - Headers: `Content-Type: application/json`
  - Body: JSON stringified preferences object
  - Timeout: 30 seconds (AbortController)
  - Zod schema validation for response
- Response time tracking: Records start/end time for analytics

d. Response Handling

- Parses JSON response (validated via Zod schema)
- Extracts results array, metadata (counts, thresholds, request_id)
- Updates React state for rendering
- Tracks results view event with `useResultsTracking` hook

3. Backend: API Endpoint

Location: `backend/app/api/v2/routes.py` → `POST /api/v2/recommendations`

a. Request Reception

- Flask route handler receives POST request
- Parses JSON body into preferences dict
- Validates required fields (optional - most are optional)
- Generates request ID (UUID) for logging

b. Recommendation Service Call

- Creates `serialize_occurrence()` function using Pydantic `TripOccurrenceSchema`
- Calls `get_recommendations(preferences, serialize_occurrence)`
- Passes serialization function to format results using Pydantic schemas

4. Backend: Recommendation Service Processing

Location: `backend/app/services/recommendation/` (package)

The recommendation service has been restructured into a modular package:

Package Structure:

- `engine.py` - Main orchestration (`get_recommendations()` function)
- `scoring.py` - Scoring algorithm implementation
- `filters.py` - Query building and filtering logic
- `relaxed_search.py` - Relaxed search expansion logic
- `constants.py` - Configuration and thresholds
- `context.py` - Preference parsing and normalization

The `get_recommendations()` function (in `engine.py`) orchestrates the pipeline:

a. Query Building (`filters.py`)

- Builds base query with eager loading (joinedload/selectinload)
- Applies geographic filters (countries/continents)
- Applies trip type filter (hard filter)
- Applies date filters (year/month, future dates only)
- Applies status filters (excludes Cancelled/Full)
- Applies difficulty filter (±1 tolerance)
- Applies budget filter (up to 30% over budget)

b. Primary Search (`engine.py` + `scoring.py`)

- Loads all candidates matching hard filters
- Scores each trip using scoring algorithm from `scoring.py`
- Scoring factors (configurable via `constants.py`):
  - Base score: 30 points (configurable)
  - Theme matching: +25 (2+ themes), +12 (1 theme), -15 (none)
  - Difficulty: +15 (exact match)
  - Duration: +12 (ideal), +8 (good, ±4 days)
  - Budget: +12 (within), +8 (110%), +5 (120%)
  - Status bonuses: +7 (Guaranteed), +15 (Last Places)
  - Departing soon: +7 (within 30 days)
  - Geography: +15 (direct country), +5 (continent)
- Sorts by score (descending), then date (ascending)
- Returns top 10 results

c. Relaxed Search (`relaxed_search.py`) (if primary results < 6)

- Expands filters:
  - Geography: Same continent if specific countries selected
  - Trip type: No filter (all types with -10 penalty)
  - Date: ±2 months from selected date
  - Difficulty: ±2 levels (instead of ±1)
  - Budget: 50% over (instead of 30%)
- Applies relaxed penalty (configurable, default -15 points)
- Scores and sorts relaxed candidates using same scoring algorithm
- Adds needed results to reach 10 total

5. Backend: Response Formatting

Location: `backend/app/api/v2/routes.py` → `serialize_occurrence()` function

Converts `TripOccurrence` objects to frontend-compatible format:

- Uses Pydantic `TripOccurrenceSchema` for serialization
- Automatically converts snake_case to camelCase via `model_dump(by_alias=True)`
- Combines template data (title, description, difficulty) with occurrence data (dates, price, status)
- Includes related entities (country, guide, trip type, tags) via eager loading
- Adds match score and match details (from scoring algorithm)
- Marks relaxed results with `is_relaxed: true` (added by recommendation service)
- Removes unnecessary fields (e.g., template countries array) for performance

6. Backend: Logging

Location: `backend/recommender/logging.py`

After returning results, logs the request:

- Request ID (UUID)
- Preferences (full JSON)
- Results (trip IDs, scores)
- Metrics (response time, candidate count, score statistics)
- Search type classification (`exploration` vs `focused_search`)

Storage: `recommendation_requests` table in PostgreSQL

7. Frontend: Results Display & Event Tracking

Location: `frontend/src/app/search/results/page.tsx`

a. Results Rendering

- Maps results array to trip cards
- Displays match scores with color coding (turquoise/orange/red)
- Shows trip details: title, description, dates, price, guide, status
- Separates primary and relaxed results with visual divider

b. Event Tracking

- **Page View**: Tracked via `usePageView('search_results')` hook
- **Results View**: Tracked via `useResultsTracking()` hook with:
  - Result count, primary/relaxed counts
  - Top score, response time
  - Recommendation request ID (for correlation)
- **Impressions**: Tracked via `useImpressionTracking()` hook when trip cards enter viewport (50% visible)
- **Scroll Depth**: Tracked at 25%, 50%, 75%, 100% thresholds
- **Clicks**: Tracked via `trackTripClick()` when user clicks a trip card
  - Includes position, score, source (`search_results` vs `relaxed_results`)
  - Flushes events before navigation to trip detail page

Event Tracking Pipeline

The event tracking pipeline captures user interactions for analytics and personalization.

Flow Overview

```
User Action (Frontend UI)
    ↓
React Event Handler (onClick, onChange, etc.)
    ↓
Tracking Function Call (trackEvent, trackPageView, etc.)
    ↓
Event Queue (tracking.service.ts)
    ↓
Batch Processing (every 5s or 10 events)
    ↓
POST /api/events/batch
    ↓
Event Service (events.py)
    ↓
Validation & User Resolution
    ↓
Database Storage (events table)
    ↓
Real-time Updates (sessions, users, trip_interactions)
```

Detailed Steps

1. Frontend: User Interaction

Location: Various React components (`frontend/src/app/**/*.tsx`)

User performs an action:

- **Clicks**: Search button, trip card, filter option, etc.
- **Changes**: Filter value, sort option
- **Views**: Page loads, trip card enters viewport
- **Scrolls**: Through results page
- **Navigates**: Between pages

2. Frontend: Event Handler Execution

Location: React component event handlers

Component event handlers call tracking functions:

- `handleSearch()` → `trackSearchSubmit(preferences, searchType)`
- `onClick={() => handleTripClick(...)}` → `trackTripClick(tripId, position, score, source)`
- `useEffect(() => {...}, [])` → `usePageView('page_name')`
- `useEffect(() => {...}, [isVisible])` → `useImpressionTracking(tripId, position, score, source)`

3. Frontend: Tracking Service Processing

Location: `frontend/src/services/tracking.service.ts` , `frontend/src/hooks/useTracking.ts`

Note: The tracking service remains in `frontend/src/services/` while API client functions are in `frontend/src/api/events.ts` . The tracking service uses the API client for network requests.

a. Identity Management

- Gets or creates `anonymous_id` from localStorage (persists across sessions)
- Gets or creates `session_id` from localStorage (expires after 30 minutes)
- Detects device type from `window.innerWidth` (not user-agent):
  - Mobile: < 768px
  - Tablet: 768px - 1023px
  - Desktop: ≥ 1024px

b. Session Initialization

- On first event, calls `initializeSession()` :
  - POST `/api/session/start` with device_type, referrer, user_agent, IP
  - Backend creates/updates session record
  - Links session to user (anonymous or registered)

c. Event Creation Each tracking function creates an event object:

```
{
  event_type: 'click_trip' | 'search_submit' | 'page_view' | etc.,
  trip_id?: number,
  recommendation_request_id?: string,
  source?: 'search_results' | 'relaxed_results' | 'homepage' | etc.,
  metadata?: { duration_seconds: 45, filter_name: 'budget', ... },
  position?: number,
  score?: number,
  client_timestamp: '2025-01-01T12:00:00Z',
  page_url: '/search/results',
  referrer: 'https://google.com/...'
}
```

d. Event Queuing

- Events are added to `eventQueue` array
- Queue is processed in batches for efficiency:
  - **Batch size**: 10 events
  - **Batch interval**: 5 seconds
  - **Immediate flush**: If queue reaches 10 events

e. Batch Sending When batch is ready:

- Gets current `anonymous_id` and `session_id`
- Optionally gets authenticated user email (if Supabase auth available)
- Adds identity to each event
- Calls `trackEventsBatch()` from `frontend/src/api/events.ts`
- Which POSTs to `/api/events/batch` with array of events
- Uses `keepalive: true` for reliable delivery during page unload

f. Page Unload Handling

- `beforeunload` and `pagehide` event listeners call `flushPendingEvents()`
- Uses `navigator.sendBeacon()` for reliable delivery during page unload
- Ensures no events are lost when user navigates away

4. Backend: API Endpoint Reception

Location: `backend/app/api/events/routes.py` → `POST /api/events/batch`

Receives batch of events:

- Validates JSON payload
- Checks batch size (max 100 events)
- Processes each event sequentially

2. Session Management

Location: `backend/app/api/events/routes.py` → `POST /api/session/start`

Before tracking events, frontend initializes session:

- Creates/resumes session with 30-minute timeout
- Captures device type from frontend (not user-agent parsing)
- Captures referrer, user agent, IP address
- Links to user (anonymous or registered)

3. Event Service Processing

Location: `backend/app/services/events.py`

a. User Resolution

- Priority 1: Find by Supabase user ID (if authenticated)
- Priority 2: Find by anonymous_id
- Priority 3: Find by email
- Create new user if not found

b. Event Validation

- Validates required fields (event_type, session_id, anonymous_id)
- Validates event type against `VALID_EVENT_TYPES`
- Validates UUID formats
- Validates source (if present) against `VALID_SOURCES`
- Validates trip_id (positive integer)

c. Event Storage

- Creates `Event` record with:
  - Links to `event_types` table (3NF schema)
  - User ID, session ID, anonymous ID
  - Trip ID, recommendation request ID (links to Phase 0)
  - Event data (metadata as JSONB)
  - Position, score, timestamps

d. Real-time Updates

- Updates session counters:
  - `search_count` (on search_submit)
  - `click_count` (on click_trip)
  - `save_count` (on save_trip)
  - `contact_count` (on contact_whatsapp/phone)
- Updates user counters:

    - `total_searches` , `total_clicks`
    - `last_seen_at`
- Updates trip interactions:
    - `impression_count` , `click_count` , `save_count`
    - `whatsapp_contact_count` , `phone_contact_count`
    - `booking_start_count`
    - `total_dwell_time_seconds` , `avg_dwell_time_seconds`
    - Computed rates: CTR, save_rate, contact_rate

4. Batch Processing

Location: `backend/app/api/events/routes.py` → `POST /api/events/batch`

For efficiency, frontend can batch events (max 100):

- Processes events sequentially
- Validates each event
- Tracks valid events
- Returns summary with processed count and errors

Analytics and Metrics Pipeline

The analytics pipeline aggregates recommendation and event data for insights.

Flow Overview

```
Recommendation Requests (recommendation_requests table)
    ↓
Events (events table)
    ↓
Metrics Aggregator (metrics.py)
    ↓
Daily Aggregation
    ↓
API Endpoints (/api/metrics)
    ↓
Dashboard/Reports
```

Detailed Steps

1. Data Collection

Location: `backend/recommender/logging.py` , `backend/app/services/events.py`

Data is collected in real-time:

- **Recommendation Requests**: Logged after each recommendation call
- **Events**: Tracked as users interact with the platform

2. Metrics Aggregation

Location: `backend/recommender/metrics.py`

The `MetricsAggregator` class computes metrics:

a. Daily Metrics (`aggregate_daily_metrics()`)

- Total requests, unique sessions
- Response time statistics (avg, p50, p95, p99, max)
- Score statistics (avg top score)
- Result quality metrics:
  - Relaxed trigger rate
  - No results rate
  - Low score rate (< 50)
- Filter usage:
  - Searches with country/continent/type/themes/budget/dates

b. Current Metrics (`get_current_metrics()`)

- Summary for last N days (default 7)
- Aggregated across date range
- Key performance indicators

c. Top Searches (`get_top_searches()`)

- Most common continents
- Most common trip types
- Most common themes (future)
- Most common countries (future)

3. API Endpoints

Location: `backend/app/api/analytics/routes.py`

GET /api/metrics

- Returns current metrics summary (last 7 days by default)
- Query param: `days` (1-90)

GET /api/metrics/daily

- Returns daily breakdown for date range
- Query params: `start` (YYYY-MM-DD), `end` (YYYY-MM-DD)
- Max range: 90 days

GET /api/metrics/top-searches

- Returns top search patterns
- Query params: `days` (default 7), `limit` (default 10)

4. Evaluation Pipeline

Location: `backend/recommender/evaluation.py`

Automated testing of recommendation quality:

- Loads evaluation scenarios from database
- Runs recommendations for each scenario
- Validates results (count, score thresholds, specific trip requirements)
- Generates pass/fail report

POST /api/evaluation/run

- Runs all scenarios or filtered subset
- Returns evaluation report

GET /api/evaluation/scenarios

- Lists available scenarios without running them

Resource Data Pipeline

The resource pipeline serves reference data (countries, guides, trip types, tags) to the frontend.

Flow Overview

```
Frontend Component Mounts
    ↓
DataStore Provider Initialization
    ↓
API Calls (GET /api/locations, /api/trip-types, /api/tags)
    ↓
Backend Resources API (resources/routes.py)
    ↓
Database Query (Reference Tables)
    ↓
Response to Frontend
    ↓
React State Update (dataStore.tsx)
    ↓
UI Components Render with Data
```

Detailed Steps

1. Frontend: Component Initialization

Location: `frontend/src/app/search/page.tsx`

a. Component Mount

- Search page component mounts
- `useEffect` hooks trigger data fetching:
  - `fetchCountries()` → calls `getLocations()` from `frontend/src/api/resources.ts`
  - `fetchTypesAndTags()` → calls `getTripTypes()` and `getTags()` from `frontend/src/api/resources.ts`

b. API Service Calls Location: `frontend/src/api/` (modular API structure)

The frontend API has been restructured into a modular architecture:

API Structure:

- `client.ts` - Core API utilities (`apiFetch()` wrapper, authentication, error handling)
- `resources.ts` - Resource endpoints (`getLocations()` , `getTripTypes()` , `getTags()` , `getGuides()`)
- `v2.ts` - V2 API endpoints (`getRecommendations()` , `getTemplates()` , `getOccurrences()`)
- `events.ts` - Event tracking endpoints (`trackEventsBatch()` , `startSession()`)

- `analytics.ts` - Analytics endpoints (`getMetrics()` , `getDailyMetrics()`)
- `system.ts` - System endpoints (`healthCheck()`)
- `types.ts` - TypeScript type definitions
- `index.ts` - Centralized exports

Each function uses `apiFetch()` wrapper from `client.ts` :

- Adds authentication headers (if user logged in)
- Handles retries for cold starts (network errors)
- 30-second timeout with AbortController
- Returns standardized `ApiResponse<T>` format
- Zod schema validation for runtime type checking

c. Data Mapping

- Maps backend response to frontend types
- Handles field name variations (`name_he` vs `nameHe`)
- Filters data if needed (e.g., only theme tags)

2. Backend: Resource Endpoints

Location: `backend/app/api/resources/routes.py`

GET /api/locations

- Returns all countries and continents
- Used for search dropdown
- Includes Hebrew names
- No filtering (shows all countries in database)

GET /api/countries

- Returns all countries (excludes Antarctica)
- Optional filter: `continent` query param

GET /api/countries/:id

- Returns specific country details

GET /api/guides

- Returns all active guides
- Used for guide selection/filtering

GET /api/trip-types

- Returns all trip types (trip styles)
- Used for trip type filter

GET /api/tags

- Returns all theme tags (trip interests)
- Used for theme selection
- Note: After V2 migration, only theme tags (category column removed)

3. Backend: Database Query

Location: `backend/app/core/database.py` → SQLAlchemy ORM

- Queries reference tables: `countries` , `guides` , `trip_types` , `tags`

- Uses eager loading for related data
- Filters active records (`is_active == True` where applicable)
- Orders results (alphabetically by name)

4. Frontend: Response Handling

Location: `frontend/src/app/search/page.tsx`

a. State Update

- Updates React state with fetched data:
  - `setCountries(mappedCountries)`
  - `setTripTypes(mappedTypes)`
  - `setThemeTags(mappedTags)`
- Clears loading states
- Clears error states on success

b. Error Handling

- Network errors trigger retry logic (for cold starts)
- Shows error UI if retries fail
- Provides retry button for user-initiated retry

5. Frontend: Data Usage in UI

Location: Various React components

a. Search Form Dropdowns

- Countries dropdown: Filtered by search input, grouped by continent
- Continents dropdown: Static list with Hebrew names
- Trip types: Grid of selectable circles with icons
- Theme tags: Grid of selectable circles (max 3) with icons

b. Data Display

- Country names: Display Hebrew names (`nameHe`) in UI
- Trip type names: Display Hebrew names in badges
- Guide names: Display in trip cards and detail pages
- Continent names: Display in location selection

c. DataStore Context (Alternative Pattern) Location: `frontend/src/lib/dataStore.tsx`

Some components use centralized DataStore:

- `DataStoreProvider` wraps app (optional)
- Provides hooks: `useCountries()` , `useTripTypes()` , `useThemeTags()`
- Caches data in React context
- Reduces redundant API calls across components

Authentication Pipeline

The authentication pipeline handles user registration and login via Supabase OAuth.

Flow Overview

```
User Clicks Login Button (Frontend)
    ↓
Supabase Client Sign-In Initiation
    ↓
Redirect to OAuth Provider (Google, etc.)
    ↓
User Authenticates with Provider
    ↓
OAuth Callback to /auth/callback
    ↓
Session Extraction from URL Hash
    ↓
Supabase Session Storage (localStorage)
    ↓
User Identification API Call (/api/user/identify)
    ↓
JWT Token in API Requests (Authorization Header)
```

Detailed Steps

1. Frontend: Login Initiation

Location: `frontend/src/app/auth/page.tsx`

a. User Action

- User clicks "התחבר" (Login) button
- Component calls `supabase.auth.signInWithOAuth({ provider: 'google' })`

b. OAuth Redirect

- Supabase client redirects to Google OAuth consent screen
- User grants permissions
- Google redirects back to app with authorization code

2. Frontend: OAuth Callback Handling

Location: `frontend/src/app/auth/callback/page.tsx`

a. URL Hash Parsing

- Callback page extracts session from URL hash:
  - `access_token` : JWT token
  - `refresh_token` : Refresh token
  - `expires_in` : Token expiration
  - `user` : User object with email, ID, metadata

b. Session Storage

- Stores session in Supabase client (localStorage)
- Session persists across page reloads
- Client automatically refreshes tokens when expired

c. User Identification

- Calls `/api/user/identify` endpoint:

    - Links Supabase user ID to tracking system
    - Associates anonymous_id with registered user
    - Enables cross-device tracking

d. Navigation

- Redirects to original page or `/search`
- User is now authenticated

2. Callback Processing

Location: `frontend/src/app/auth/callback/page.tsx`

- Extracts session from URL hash
- Stores session in Supabase client
- Links Supabase user to tracking system:
    - Calls `/api/user/identify` with email
    - Links anonymous_id to registered user

3. Frontend: JWT Token in API Requests

Location: `frontend/src/api/client.ts` → `getAuthHeaders()`

a. Token Retrieval

- `getAuthHeaders()` function called before each API request
- Gets access token from Supabase client: `await getAccessToken()`
- Returns `{ 'Authorization': 'Bearer <token>' }` if token exists
- Returns empty object if no token (unauthenticated requests)

b. Token Inclusion

- All API requests include auth headers automatically
- `apiFetch()` wrapper (in `client.ts`) adds headers to every request
- All API modules (`resources.ts`, `v2.ts`, `events.ts`, etc.) use `apiFetch()` for consistent authentication
- Backend can identify authenticated users

4. Backend: API Authentication

Location: `backend/app/core/auth.py`

Protected endpoints verify JWT:

- Extracts token from `Authorization` header
- Verifies token signature with Supabase public key
- Validates token expiration
- Extracts user info (email, user ID from 'sub' claim)
- Passes to route handlers via `get_current_user()`
- Returns `None` if token invalid or missing (allows anonymous access)

5. Backend: User Resolution in Events

Location: `backend/app/services/events.py`

When events are tracked:

- Checks for authenticated user via `get_current_user()` (from JWT)

- Priority order for user resolution:
    1. Supabase user ID (from JWT 'sub' claim)
    2. Email (from JWT or event payload)
    3. Anonymous ID (from event payload)
- Links events to registered user if authenticated
- Enables cross-device tracking (same user across devices)
- Updates user's `last_seen_at` timestamp

Database Architecture

Core Tables

Trip Data (V2 Schema)

- **companies**: Trip providers
- **trip_templates**: The "what" of trips (description, pricing, difficulty)
- **trip_occurrences**: The "when" of trips (dates, guide, availability)
- **trip_template_tags**: Many-to-many (templates ↔ themes)
- **trip_template_countries**: Many-to-many (templates ↔ countries)

Reference Data

- **countries**: Country information with Hebrew names
- **guides**: Tour guide information
- **trip_types**: Trip style categories
- **tags**: Theme tags (interests)

Analytics Tables

- **recommendation_requests**: Logged recommendation calls
- **users**: Anonymous and registered users
- **sessions**: Browser sessions with device info
- **events**: User interaction events
- **event_types**: Event type reference (3NF)
- **trip_interactions**: Aggregated trip engagement metrics

Data Relationships

```
TripTemplate (1) ──→ (N) TripOccurrence
TripTemplate (N) ──→ (N) Tag (via trip_template_tags)
TripTemplate (N) ──→ (N) Country (via trip_template_countries)
TripTemplate (1) ──→ (1) Company
TripTemplate (1) ──→ (1) TripType
TripTemplate (1) ──→ (1) Country (primary_country)

TripOccurrence (1) ──→ (1) Guide
TripOccurrence (1) ──→ (1) TripTemplate

User (1) ──→ (N) Session
User (1) ──→ (N) Event
Session (1) ──→ (N) Event
Event (1) ──→ (1) EventType
Event (N) ──→ (1) TripOccurrence (optional)
```

```
TripOccurrence (1) ──→ (1) TripInteraction
```

Data Flow Patterns

Write Pattern

1. Frontend sends request to API
2. API validates input
3. Database transaction begins
4. Data is written/updated
5. Related counters updated (sessions, users, trip_interactions)
6. Transaction commits
7. Response returned to frontend

Read Pattern

1. Frontend requests data
2. API builds optimized query (eager loading)
3. Database query executed
4. Results formatted (to_dict methods)
5. Response returned to frontend

Analytics Pattern

1. Events/recommendations logged in real-time
2. Batch aggregation jobs (future: scheduled daily)
3. Metrics computed from aggregated data
4. API serves metrics to dashboard

Performance Considerations

Query Optimization

- **Eager Loading**: Uses `joinedload` and `selectinload` to avoid N+1 queries
- **Indexes**: Database indexes on foreign keys and frequently filtered columns
- **Connection Pooling**: SQLAlchemy connection pool (5-10 connections)

Caching Strategy

- **Frontend**: Resource data cached in React context
- **Backend**: No caching currently (future: Redis for frequently accessed data)

Batch Processing

- **Events**: Supports batch upload (max 100 events per request)
- **Metrics**: Aggregated on-demand (future: scheduled daily aggregation)

Scalability

- **Read Replicas**: Can add read replicas for analytics queries
- **Partitioning**: `recommendation_requests` and `events` tables can be partitioned by date
- **Archiving**: Old events/recommendations can be archived to separate tables

---

Future Enhancements

1. **Scheduled Aggregation**: Daily batch jobs for metrics computation

2. **Real-time Analytics**: WebSocket updates for live dashboard
3. **Machine Learning**: Use event data to improve recommendation scoring
4. **A/B Testing**: Track recommendation algorithm variants
5. **Personalization**: Use user event history to personalize recommendations
6. **Caching Layer**: Redis for frequently accessed data
7. **Data Warehouse**: ETL pipeline to data warehouse for advanced analytics

---

Related Documentation

- API Reference
- Database Schema
- Frontend Overview
- Supabase OAuth Setup