API Endpoints Flow with Supabase Functions

This document provides a comprehensive explanation of how API endpoints interact with Supabase database functions in the AI Research Assistant application. It details the complete flow from HTTP requests to database operations.

Table of Contents

Architecture Overview

The application follows a 3-tier architecture with clear separation of concerns:

```
Frontend (Next.js) → Express DB Server → Supabase RPC Functions →
PostgreSQL Database
     ↓                    ↓                      ↓
↓
User Interface     HTTP API Endpoints    Business Logic Functions   Data
Storage
```

PROF

Key Components

1. **Frontend (Port 3000)**: Next.js application handling user interface
2. **Express DB Server (Port 3001)**: API layer that exclusively handles database operations via Supabase RPC
3. **FastAPI AI Server (Port 8000)**: AI/ML operations with NO direct database access
4. **Supabase**: Database with RPC functions for business logic
5. **PostgreSQL**: Underlying database storage

Request Flow Pattern

Every database operation follows this consistent pattern:

```
1. Client Request → 2. Express Route Handler → 3. RPC Function Call → 4.
Database Operation → 5. Response
```

Standard Flow Example

```
// 1. Client Request
POST /api/groups
{
  "name": "Research Group",
  "description": "AI Research Team",
  "is_public": true
}

// 2. Express Route Handler (/routes/groups.js)
router.post('/', async (req, res, next) => {
  try {
    const { name, created_by, description, is_public } = req.body;
    const supabase = req.app.locals.supabase;

    // 3. RPC Function Call
    const result = await executeRPC(supabase, 'create_group', {
      p_name: name,
      p_created_by: created_by,
      p_description: description,
      p_is_public: is_public
    });

    // 5. Response
    res.json(result);
  } catch (error) {
    next(error);
  }
});

// 4. Database Operation (Supabase RPC Function)
CREATE OR REPLACE FUNCTION public.create_group(
  p_name text,
  p_created_by integer,
  p_description text DEFAULT ''::text,
  p_is_public boolean DEFAULT false
)
RETURNS json
```

Authentication Flow

Current User Authentication

PROF

Endpoint: `GET /api/auth/me`
Flow:

1. **Client Request**: JWT token in Authorization header
2. **Express Handler**: `/routes/auth.js`
3. **RPC Function**: `get_current_user_auth()`
4. **Database Query**:

```
SELECT user_id, email, first_name, last_name, is_active,
created_at, auth_id
FROM users
WHERE auth_user_id = auth.uid()
```

5. **Response**: User profile data

Profile Update Flow

Endpoint: `PUT /api/auth/me`
Flow:

1. **Validation**: Check required fields and authentication
2. **RPC Call**: `update_user(p_user_id, p_email, p_first_name, p_last_name)`
3. **Database Operations**:
    - Update user record
    - Trigger `update_updated_at_column` for timestamp
4. **Response**: Updated user profile

Profile Sync Flow

Endpoint: `POST /api/auth/sync-profile`
Flow:

PROF

1. **Auth Check**: Verify JWT token and extract user data
2. **Profile Creation/Update**: Call `syncUserProfile()` helper
3. **RPC Operations**:
    - `get_user_by_auth_id(auth_id)` - Check existing profile
    - `create_user()` or `update_user()` - Create/update as needed
4. **Response**: Synchronized profile data

User Management Flow

Get All Users

Endpoint: `GET /api/users`
Flow:

1. **Request**: Optional pagination parameters (`limit`, `offset`)
2. **RPC Call**: `get_all_users()`

3. **Database Query**:

```
SELECT user_id as id, auth_user_id as auth_id,
       COALESCE(first_name || ' ' || last_name, email) as name,
       email, is_active
FROM users
ORDER BY created_at DESC
```

4. **Response**: Array of user objects

User Creation Flow

Endpoint: `POST /api/users`
Flow:

1. **Validation**: Email format and required fields
2. **RPC Call**: `create_user(p_email, p_first_name, p_last_name)`
3. **Database Operations**:
   - Check email uniqueness
   - Insert new user record
   - Set default availability to 'available'
   - Trigger `handle_new_user` for additional setup
4. **Response**: Created user with computed name field

Group Management Flow

Group Creation Flow

Endpoint: `POST /api/groups`
Flow:

PROF

1. **Authentication**: Verify user can create groups (user_id >= 2)
2. **Validation**: Group name and description
3. **RPC Call**: `create_group(p_name, p_created_by, p_description, p_is_public)`
4. **Database Operations**:

```
-- Insert group
INSERT INTO groups (name, description, created_by, is_public)
VALUES (p_name, p_description, p_created_by, p_is_public)

-- Auto-generate invite code via trigger
TRIGGER: set_group_invite_code()

-- Add creator as admin
INSERT INTO group_participants (group_id, user_id, role)
VALUES (v_group_id, p_created_by, 'admin')
```

5. **Response**: Group details with invite code

Join Group Flow

Endpoint: `POST /api/groups/:id/join`
Flow:

1. **Request**: Group ID and invite code
2. **RPC Call**: `join_group_by_invite_code(p_invite_code, p_user_id)`
3. **Database Operations**:
   - Validate invite code exists
   - Check if user already member
   - Add user to group_participants
   - Update group member count
4. **Response**: Success confirmation with group details

Group Members Retrieval

Endpoint: `GET /api/groups/:id/members`
Flow:

1. **Access Check**: `can_user_access_group(p_user_id, p_group_id)`
2. **RPC Call**: `get_group_members_detailed(p_group_id)`
3. **Database Query**:

```
SELECT u.user_id, u.first_name, u.last_name, u.email,
       gp.role, gp.joined_at, u.is_active
FROM group_participants gp
JOIN users u ON gp.user_id = u.user_id
WHERE gp.group_id = p_group_id
```

4. **Response**: Detailed member list with roles

PROF

Session Management Flow

Session Creation Flow

Endpoint: `POST /api/sessions`
Flow:

1. **Authentication**: Verify user permissions
2. **RPC Call**: `create_session(p_title, p_user_id, p_group_id)`
3. **Database Operations**:

```
-- Create session
INSERT INTO sessions (title, description, created_by, group_id,
status)
```

```
VALUES (p_title, p_description, p_created_by, p_group_id, 'active')

-- Auto-add creator as participant
INSERT INTO session_participants (session_id, user_id, joined_at)
VALUES (v_session_id, p_created_by, NOW())
```

4. **Response**: Session details with participant count

Join Session Flow

Endpoint: `POST /api/sessions/:id/join`
Flow:

1. **Access Validation**: `can_user_access_session(p_user_id, p_session_id)`
2. **RPC Call**: `add_session_participant(p_session_id, p_user_id)`
3. **Database Operations**:
   - Check if user already participant
   - Add to session_participants table
   - Update session participant count
4. **Response**: Updated session with new participant count

Session Participants

Endpoint: `GET /api/sessions/:id/participants`
Flow:

1. **RPC Call**: `get_session_participants(p_session_id)`
2. **Database Query**:

```
SELECT sp.session_id,
       array_agg(sp.user_id) as participant_ids,
       count(sp.user_id) as participant_count
FROM session_participants sp
WHERE sp.session_id = p_session_id
GROUP BY sp.session_id
```

3. **Response**: Array of participant IDs and count

PROF

Message System Flow

Send Message Flow

Endpoint: `POST /api/messages`
Flow:

1. **Authentication**: Verify user can access session
2. **RPC Call**: `create_message(p_session_id, p_sender_user_id, p_content)`
3. **Database Operations**:

```
-- Insert message
INSERT INTO messages (session_id, sender_id, content, message_type,
sent_at)
VALUES (p_session_id, p_sender_user_id, p_content, 'user', NOW())

-- Get sender name for response
JOIN users u ON m.sender_id = u.user_id
```

4. **Real-time**: Supabase real-time broadcasts to session participants
5. **Response**: Message with sender details

Get Session Messages

Endpoint: `GET /api/messages/session/:sessionId`
Flow:

1. **Access Check**: Verify user can access session
2. **RPC Call**: `get_session_messages(p_session_id, p_limit, p_offset)`
3. **Database Query**:

```
SELECT m.message_id, m.session_id, m.sender_id, m.content,
m.sent_at,
       COALESCE(u.first_name || ' ' || u.last_name, u.email) as
sender_name
FROM messages m
LEFT JOIN users u ON m.sender_id = u.user_id
WHERE m.session_id = p_session_id
ORDER BY m.sent_at ASC
```

4. **Response**: Chronological message list with sender info

PROF

Group Chat Messages

Endpoint: `GET /api/group-chat/:groupId/messages`
Flow:

1. **Group Access**: Verify user is group member
2. **RPC Call**: `get_group_chat_messages(p_session_id, p_limit, p_offset)`
3. **Database Operations**:
   - Find active group chat session
   - Retrieve messages with sender details
   - Apply pagination
4. **Response**: Group chat message history

Paper Management Flow

Paper Creation Flow

Endpoint: `POST /api/papers`
Flow:

1. **Validation**: Title, authors, and metadata
2. **RPC Call**: `create_paper(p_title, p_authors, p_abstract, ...)`
3. **Database Operations**:

```
INSERT INTO papers (title, authors, abstract, url, pdf_path, tags,
publish_date, journal, doi)
VALUES (p_title, p_authors, p_abstract, p_url, p_pdf_path, p_tags,
p_publish_date, p_journal, p_doi)

-- Auto-update timestamp trigger
TRIGGER: update_papers_updated_at()
```

4. **Response**: Created paper with generated ID

arXiv Paper Integration

Endpoint: `POST /api/papers/search/arxiv`
Flow:

1. **External API Call**: Query arXiv API with search parameters
2. **XML Parsing**: Convert arXiv XML response to structured data
3. **Data Transformation**: Map arXiv fields to internal schema
4. **Response**: Array of arXiv papers (not stored until explicitly saved)

Save arXiv Paper: `POST /api/papers/arxiv`
Flow:

PROF

1. **RPC Call**: `create_arxiv_paper(p_title, p_abstract, p_authors, ...)`
2. **Database Operations**:

```
INSERT INTO papers (title, abstract, authors, arxiv_id, categories,
                    published_at, source_url, pdf_url, doi,
journal_ref)
VALUES (p_title, p_abstract, p_authors, p_arxiv_id, p_categories,
...)
```

3. **Response**: Saved arXiv paper with database ID

Paper Search Flow

Endpoint: `GET /api/papers?search=query`
Flow:

1. **RPC Call**: `search_papers(p_query, p_limit, p_offset)`
2. **Database Query** (Full-text search):

```
SELECT *,
       ts_rank(to_tsvector('english', title || ' ' || abstract),
               plainto_tsquery('english', p_query)) as
relevance_score
FROM papers
WHERE to_tsvector('english', title || ' ' || abstract) @@
plainto_tsquery('english', p_query)
ORDER BY relevance_score DESC
```

3. **Response**: Ranked search results with relevance scores

Link Paper to Session

Endpoint: `POST /api/papers/sessions/:sessionId/:paperId`
Flow:

1. **Access Check**: Verify user can modify session
2. **RPC Call**: `add_paper_to_session(p_session_id, p_paper_id)`
3. **Database Operations**:

```
INSERT INTO session_papers (session_id, paper_id, added_at)
VALUES (p_session_id, p_paper_id, NOW())
ON CONFLICT (session_id, paper_id) DO NOTHING
```

4. **Response**: Success confirmation

AI Integration Flow

PROF

AI Chat Request

Endpoint: `POST /ai/chat/{session_id}` (FastAPI)
Flow:

1. **FastAPI Handler**: Receives chat request
2. **HTTP Request to Express**: Get session context

```
// FastAPI calls Express DB Server
const response = await fetch('http://express-db-
server:3001/api/sessions/' + session_id);
```

3. **Express Response**: Session details and participants
4. **AI Processing**: Generate response using Groq/OpenAI

5. **Metadata Logging**: FastAPI calls Express to log AI usage

```
await fetch('http://express-db-server:3001/api/ai-metadata', {
  method: 'POST',
  body: JSON.stringify({
    session_id, model_name, prompt_tokens, completion_tokens,
response_time_ms
  })
});
```

AI Metadata Tracking

Endpoint: `POST /api/ai-metadata`
Flow:

1. **RPC Call**: `create_ai_metadata(p_session_id, p_message_id, p_model_name, ...)`
2. **Database Operations**:

```
INSERT INTO ai_metadata (session_id, message_id, model_name,
prompt_tokens,
                         completion_tokens, total_tokens,
response_time_ms, metadata)
VALUES (p_session_id, p_message_id, p_model_name, p_prompt_tokens,
...)
```

3. **Analytics**: Data used for usage statistics and performance monitoring

AI Performance Analytics

Endpoint: `GET /api/ai-metadata/stats`
Flow:

PROF

1. **RPC Call**: `get_ai_usage_stats()`
2. **Database Aggregation**:

```
SELECT COUNT(*) as total_requests,
       SUM(prompt_tokens) as total_prompt_tokens,
       AVG(response_time_ms) as avg_response_time_ms,
       mode() WITHIN GROUP (ORDER BY model_name) as most_used_model
FROM ai_metadata
```

3. **Response**: Comprehensive AI usage statistics

Error Handling Flow

Standard Error Pattern

All endpoints follow consistent error handling:

```
try {
  // 1. Validation
  if (!requiredField) {
    return res.status(400).json({ error: 'Field is required', code: 400
});
  }

  // 2. RPC Function Call
  const result = await executeRPC(supabase, 'function_name', params);

  // 3. Success Response
  res.json(result);
} catch (error) {
  // 4. Error Handling
  next(error); // Passes to global error handler
}
```

Database Error Mapping

Common PostgreSQL errors are mapped to HTTP status codes:

- `23505` (Unique Violation) → `409 Conflict`
- `23514` (Check Violation) → `400 Bad Request`
- `P0002` (No Data Found) → `404 Not Found`
- `42P01` (Undefined Table) → `500 Internal Server Error`

Authentication Errors

PROF

```
// JWT Validation Failure
{
  "error": "Invalid or expired token",
  "code": 401
}

// Insufficient Permissions
{
  "error": "Access denied to resource",
  "code": 403
}
```

RAG System Flow

Document Upload and Processing

Endpoint: `POST /api/rag/upload`
Flow:

1. **File Upload**: PDF document received
2. **Text Extraction**: Convert PDF to text
3. **Chunking**: Split text into manageable chunks
4. **Embedding Generation**: Create vector embeddings
5. **Storage**: Store in vector database with metadata
6. **RPC Call**: `create_rag_document(p_session_id, p_filename, p_metadata)`

RAG-Enhanced Chat

Endpoint: `POST /ai/chat/rag/{session_id}`
Flow:

1. **Query Reception**: User question received
2. **Vector Search**: Find relevant document chunks
3. **Context Assembly**: Combine relevant chunks
4. **AI Request**: Send query + context to LLM
5. **Response Generation**: AI generates contextual answer
6. **Metadata Logging**: Track RAG usage and performance

RAG Session Management

Endpoint: `GET /api/rag/sessions/:sessionId/documents`
Flow:

1. **RPC Call**: `get_session_rag_documents(p_session_id)`
2. **Database Query**:

```
SELECT rd.document_id, rd.filename, rd.metadata, rd.created_at
FROM rag_documents rd
WHERE rd.session_id = p_session_id
ORDER BY rd.created_at DESC
```

3. **Response**: List of uploaded documents for session

PROF

Database Function Categories

Core Utility Functions

- `generate_invite_code()` - Creates unique 8-character codes
- `generate_secure_token(p_length)` - Creates secure random tokens
- `current_user_id()` - Gets authenticated user ID
- `hash_password(p_password)` - Hashes passwords with bcrypt
- `verify_password(p_password, p_hash)` - Verifies password hashes

User Management Functions

- `create_user()` - Creates new user accounts
- `get_user_by_id()` - Retrieves user details
- `get_all_users()` - Lists all users with pagination
- `update_user()` - Updates user information
- `get_current_user_auth()` - Gets authenticated user data
- `get_user_by_auth_id()` - Finds user by Supabase auth ID

Group Management Functions

- `create_group()` - Creates groups with auto-generated invite codes
- `get_all_groups()` - Lists all groups with member counts
- `join_group_by_invite_code()` - Joins groups using invite codes
- `get_group_members_detailed()` - Gets detailed member information
- `can_user_access_group()` - Checks group access permissions
- `add_group_member()` - Adds users to groups with roles
- `remove_group_member()` - Removes users from groups

Session Management Functions

- `create_session()` - Creates new chat sessions
- `get_all_sessions()` - Lists sessions with filtering
- `add_session_participant()` - Adds users to sessions
- `get_session_participants()` - Gets session participants
- `can_user_access_session()` - Checks session access
- `get_session_by_id()` - Retrieves session details

Message Functions

- `create_message()` - Creates new messages
- `get_session_messages()` - Retrieves session messages
- `update_message()` - Updates message content
- `delete_message()` - Deletes messages
- `search_messages()` - Searches messages by content
- `send_group_chat_message()` - Specialized group chat messaging

PROF

Paper Management Functions

- `create_paper()` - Creates new paper entries
- `create_arxiv_paper()` - Creates arXiv-specific papers
- `get_all_papers()` - Lists all papers
- `search_papers()` - Full-text search in papers
- `add_paper_to_session()` - Links papers to sessions
- `get_session_papers()` - Gets papers for sessions
- `search_papers_by_tags()` - Tag-based paper search

AI Metadata Functions

- `create_ai_metadata()` - Logs AI usage metadata

- `get_ai_usage_stats()` - AI usage statistics
- `get_ai_model_performance_stats()` - Performance by model
- `get_session_ai_metadata()` - AI usage per session
- `get_top_ai_usage_sessions()` - Sessions with highest AI usage

Feedback System Functions

- `create_feedback()` - Creates feedback entries
- `get_all_feedback()` - Lists feedback with filtering
- `update_feedback()` - Updates feedback
- `get_feedback_stats()` - Feedback statistics
- `get_session_feedback()` - Session-specific feedback

Permission and Security Functions

- `can_user_modify_resource()` - Checks modification permissions
- `can_user_invoke_ai()` - Checks AI invocation permissions
- `verify_user_auth()` - Verifies user authentication
- `get_user_permissions()` - Gets user permission levels

Real-time and Presence Functions

- `update_user_presence()` - Updates user online status
- `get_session_online_users()` - Gets online session users
- `cleanup_old_presence()` - Removes stale presence records

Analytics Functions

- `get_papers_stats()` - Paper usage statistics
- `get_popular_papers()` - Most used papers
- `get_recent_papers()` - Recently added papers
- `get_ai_usage_trends()` - AI usage over time
- `get_feedback_trends()` - Feedback trends analysis

PROF

Best Practices for Endpoint Development

1. Consistent RPC Function Naming

```
-- Pattern: action_resource_[modifier]
create_user()          -- Creates a user
get_user_by_id()       -- Retrieves user by ID
update_user()          -- Updates user
delete_user()          -- Deletes user
search_papers()        -- Searches papers
get_session_messages() -- Gets messages for session
```

2. Parameter Naming Convention

```
-- Use p_ prefix for parameters
CREATE FUNCTION create_group(
  p_name text,
  p_created_by integer,
  p_description text DEFAULT ''::text
)
```

3. Error Handling in RPC Functions

```
BEGIN
  -- Validation
  IF p_name IS NULL OR trim(p_name) = '' THEN
    RAISE EXCEPTION 'Name is required' USING ERRCODE = '23514';
  END IF;

  -- Business logic
  -- ...

EXCEPTION
  WHEN others THEN
    RAISE;
END;
```

4. Consistent Response Formats

```
// Success Response
{
  "data": [...],
  "pagination": {
    "limit": 20,
    "offset": 0,
    "total": 150
  }
}

// Error Response
{
  "error": "Error description",
  "code": 400,
  "details": "Additional context"
}
```

PROF

5. Security Considerations

- All RPC functions use `SECURITY DEFINER`

- Row Level Security (RLS) policies on sensitive tables
- Authentication checks in functions
- Input validation and sanitization
- Parameter type enforcement

This comprehensive flow documentation provides developers with a complete understanding of how the API endpoints interact with Supabase functions to deliver the application's functionality while maintaining security, performance, and consistency.