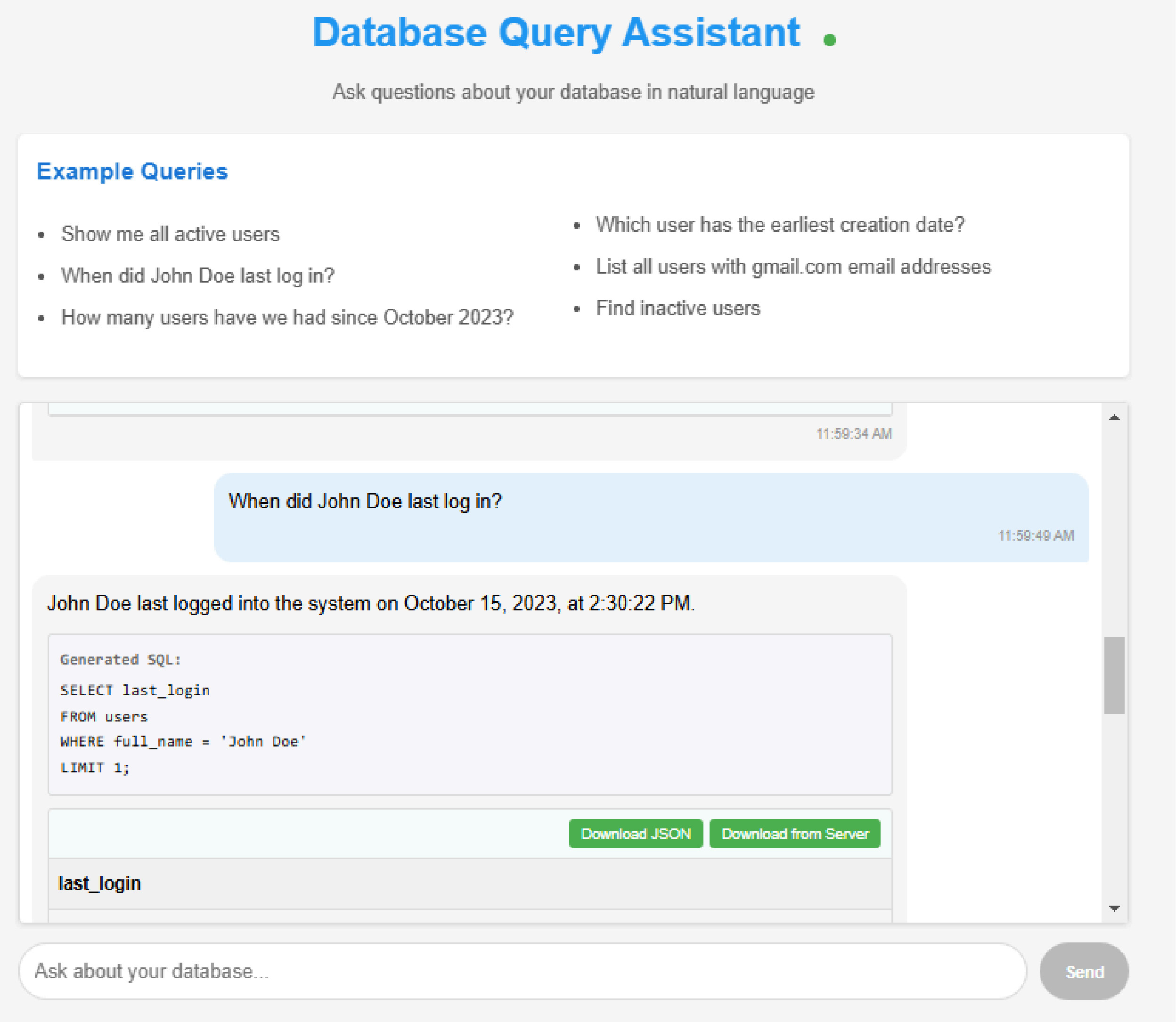
# Natural Query Interface: Project Report

By Mihai Constantinescu

## Project Overview

The RAG Chat project implements a natural language interface for database querying, allowing users to interact with database systems using plain English rather than SQL. This system bridges the gap between everyday language and structured query languages, making data exploration accessible to users without technical SQL knowledge.



Database Query Assistant

Ask questions about your database in natural language

Example Queries

- Show me all active users
- When did John Doe last log in?
- How many users have we had since October 2023?
- Which user has the earliest creation date?
- List all users with gmail.com email addresses
- Find inactive users

11:59:34 AM

When did John Doe last log in?

11:59:49 AM

John Doe last logged into the system on October 15, 2023, at 2:30:22 PM.

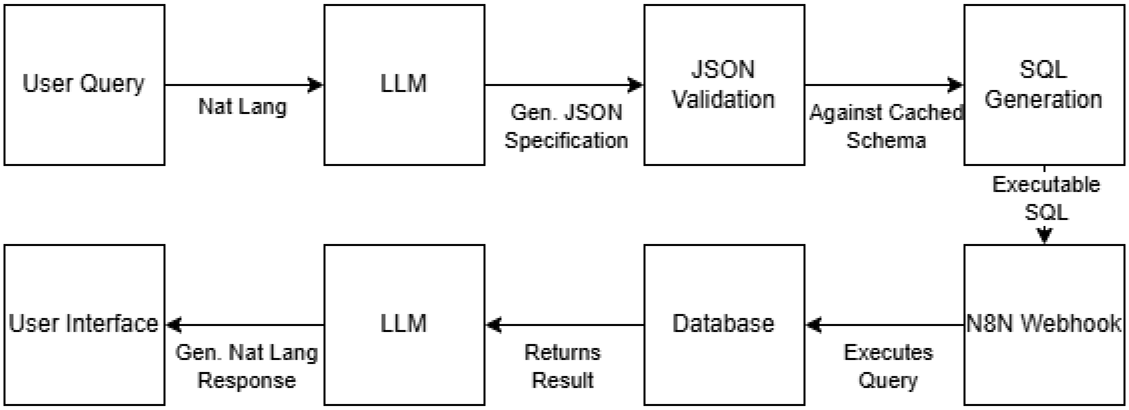
```
Generated SQL:
SELECT last_login
FROM users
WHERE full_name = 'John Doe'
LIMIT 1;
```

| last_login |
|---|

# System Architecture

The application processes queries through a multi-stage architecture. First, a Large Language Model (LLM) converts the user's natural language question and conversation context into a structured JSON specification. This JSON is validated against the cached database schema and transformed into an executable SQL query. The query is then sent to an n8n webhook, which executes it against the database. A second LLM call transforms the raw query results into a natural language response that directly answers the user's original question.

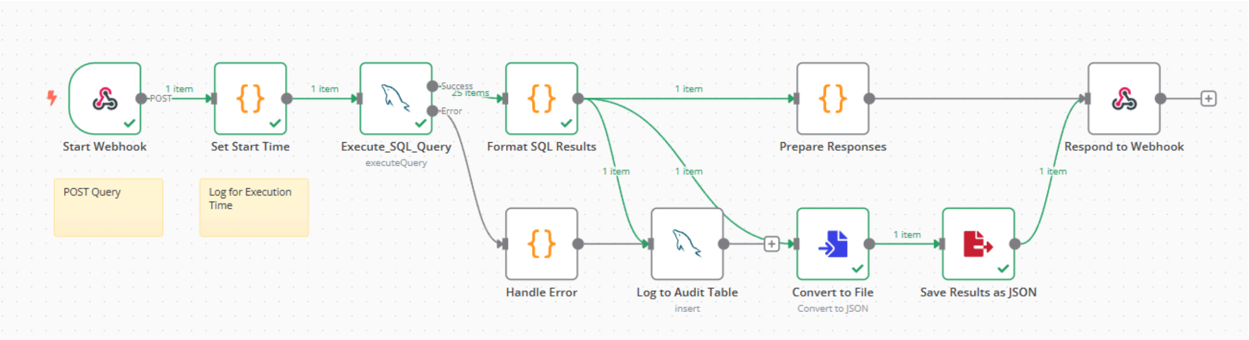
1. Users ask questions through a React UI with no need for SQL knowledge
2. On initialization, the system **dynamically** discovers and caches the database schema, incorporating it into the LLM system prompt
3. Backend processes questions using OpenAI to generate SQL based on the question, conversation context, and cached schema
4. Queries are executed via n8n workflow which handles execution, error checking, file conversion, and comprehensive logging
5. Query results are processed by a second OpenAI call that transforms technical data into natural language responses
6. The frontend receives the complete package and enables file downloads of results, with SQL queries visible only in development mode



# Core Components

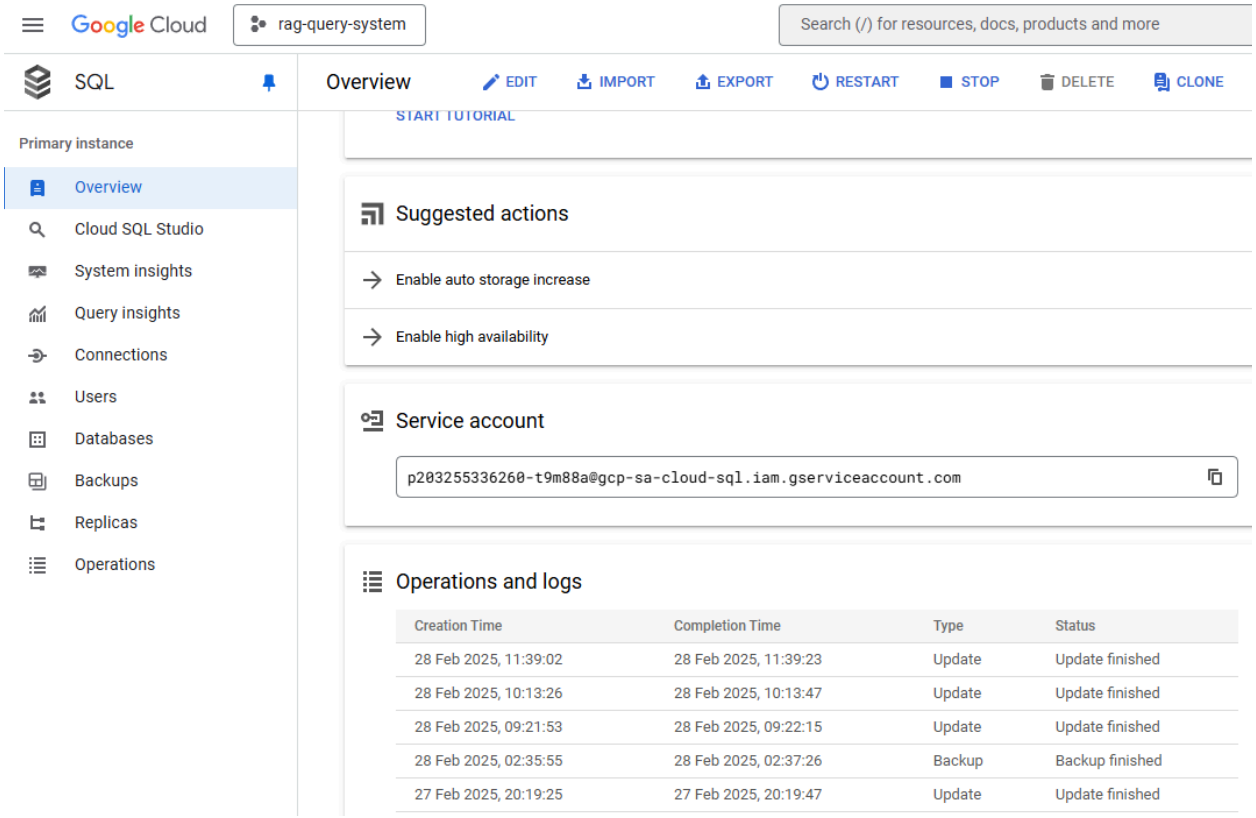
## n8n Workflow Engine

The system employs n8n to manage database operations with a focus on security and reliability. Its workflow is structured around several integrated components: a webhook endpoint receives queries, validation and execution nodes process them, comprehensive audit logging tracks activity, and JavaScript-based output formatting ensures consistency. This design separates database operations from the main application, enhancing security through an additional protective layer while allowing flexible, independent functionality.



## Database System

The application relies on MySQL as its foundation for storing data and query logs. Its implementation features two primary tables: one for user information and another for audit logs. A key capability is **dynamic schema discovery**, which allows the system to adapt seamlessly to any database structure. To ensure efficiency and security, the design incorporates connection pooling for optimized performance, query auditing for robust monitoring, and result formatting for consistent, reliable output.



Suggested actions

Enable auto storage increase

Enable high availability

Service account

p203255336260-t9m88a@gcp-sa-cloud-sql.iam.gserviceaccount.com

Operations and logs

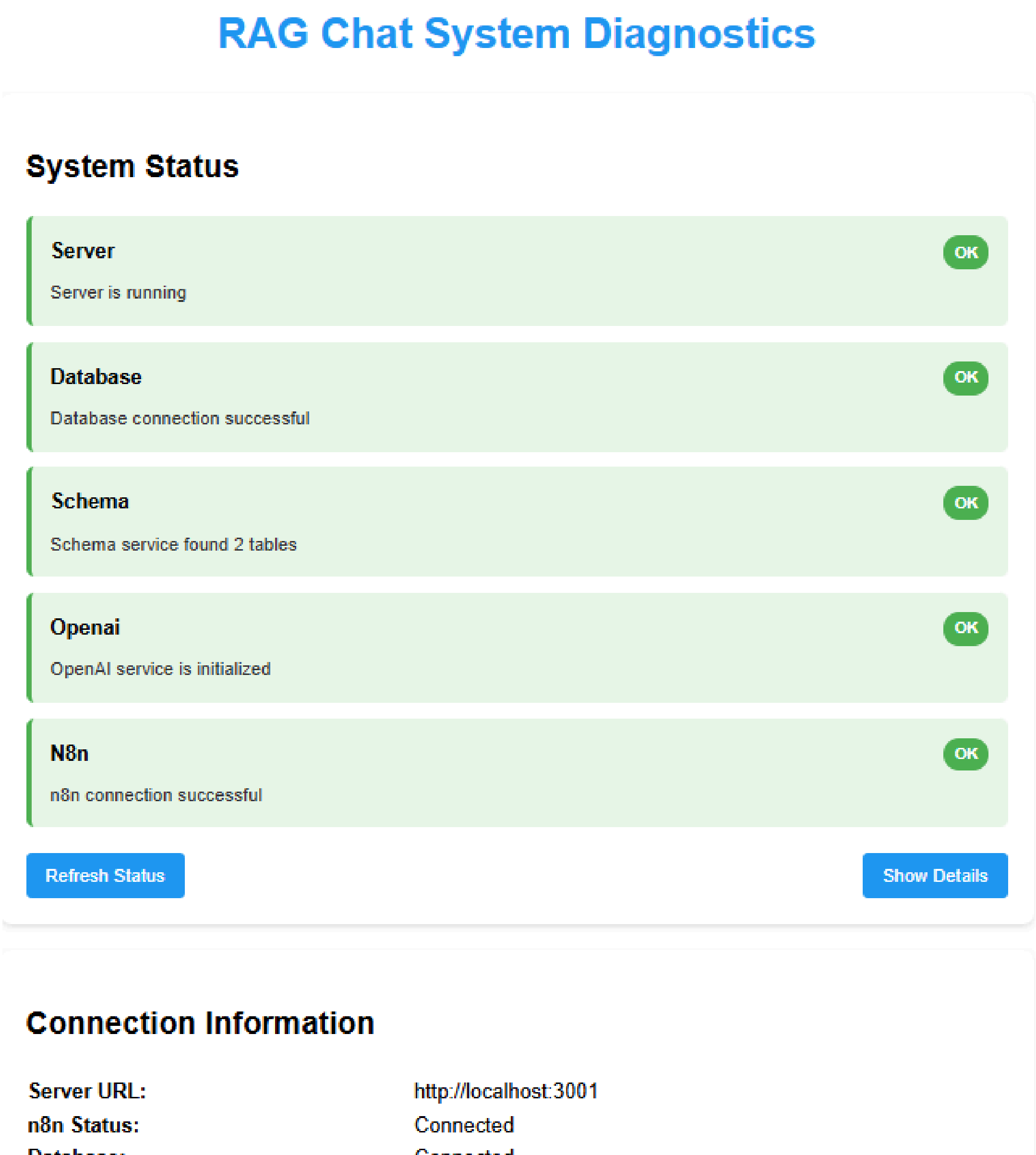
| Creation Time | Completion Time | Type | Status |
|---|---|---|---|
| 28 Feb 2025, 11:39:02 | 28 Feb 2025, 11:39:23 | Update | Update finished |
| 28 Feb 2025, 10:13:26 | 28 Feb 2025, 10:13:47 | Update | Update finished |
| 28 Feb 2025, 09:21:53 | 28 Feb 2025, 09:22:15 | Update | Update finished |
| 28 Feb 2025, 02:35:55 | 28 Feb 2025, 02:37:26 | Backup | Backup finished |
| 27 Feb 2025, 20:19:25 | 27 Feb 2025, 20:19:47 | Update | Update finished |

### Backend Implementation

Originally implemented as a Flask server, the backend has been refactored into a Node.js application using TypeScript and Express. This modular architecture improves type safety, maintainability, and developer experience. The system includes specialized services for database connection, schema discovery, OpenAI integration, n8n communication, and structured logging.

## User Interface

The application provides a primary chat-style interface with SQL display and data visualization, along with a diagnostics panel for system monitoring. These interfaces include real-time feedback, interactive examples, comprehensive error handling with troubleshooting suggestions, and data export capabilities.



RAG Chat System Diagnostics

System Status

**Server** OK
Server is running

**Database** OK
Database connection successful

**Schema** OK
Schema service found 2 tables

**Openai** OK
OpenAI service is initialized

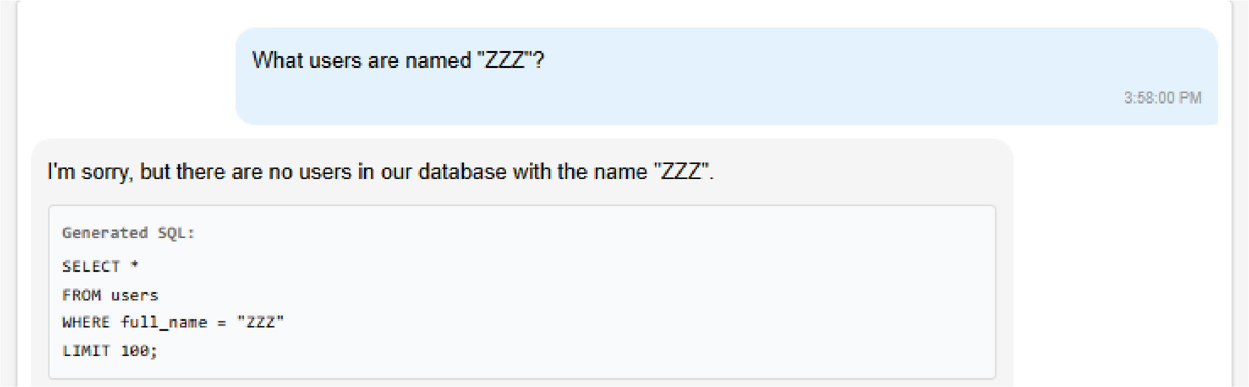
**N8n** OK
n8n connection successful

Connection Information

**Server URL:** http://localhost:3001
**n8n Status:** Connected

## Error Handling and Reliability

The system implements robust error handling at multiple levels, with input validation for all user queries and SQL security validation to prevent dangerous operations. Timeout handling, detailed error messages with troubleshooting suggestions, and comprehensive logging ensure reliability even when facing connectivity or service issues.



What users are named "ZZZ"?

3:58:00 PM

I'm sorry, but there are no users in our database with the name "ZZZ".

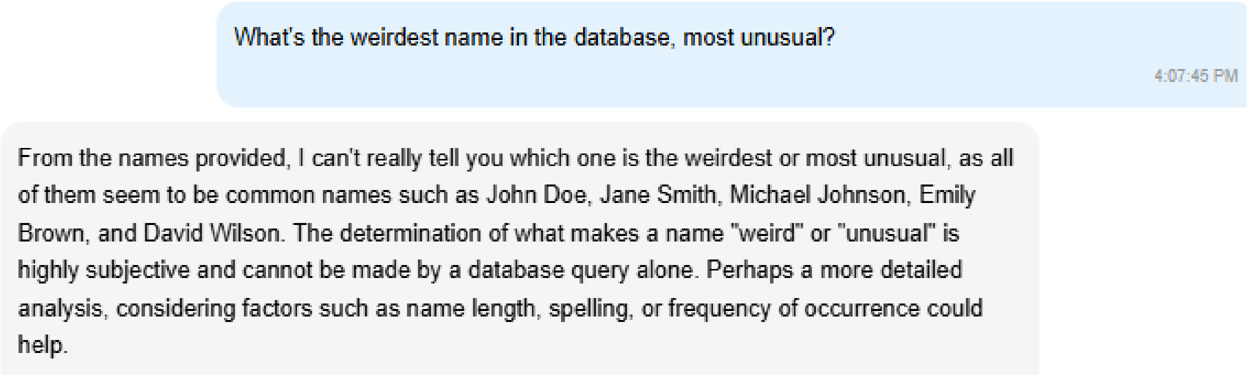
```
Generated SQL:
SELECT *
FROM users
WHERE full_name = "ZZZ"
LIMIT 100;
```

## Conclusion

The RAG Chat Natural Query Interface successfully demonstrates how modern AI can enhance database accessibility. By combining LLMs with traditional database technologies, it creates an intuitive bridge between natural language and SQL. The system's modular architecture, error handling, and thoughtful user interfaces make it a robust solution for organizations looking to democratize access to their data.

# Gallery:

## Natural Response

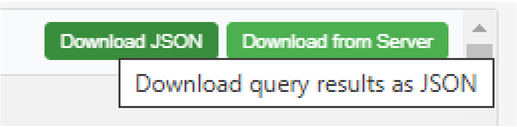


What's the weirdest name in the database, most unusual?

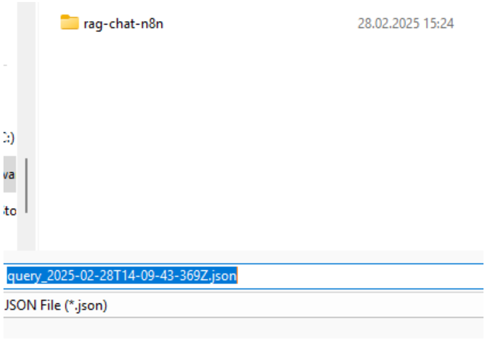
4:07:45 PM

From the names provided, I can't really tell you which one is the weirdest or most unusual, as all of them seem to be common names such as John Doe, Jane Smith, Michael Johnson, Emily Brown, and David Wilson. The determination of what makes a name "weird" or "unusual" is highly subjective and cannot be made by a database query alone. Perhaps a more detailed analysis, considering factors such as name length, spelling, or frequency of occurrence could help.

## JSON Download



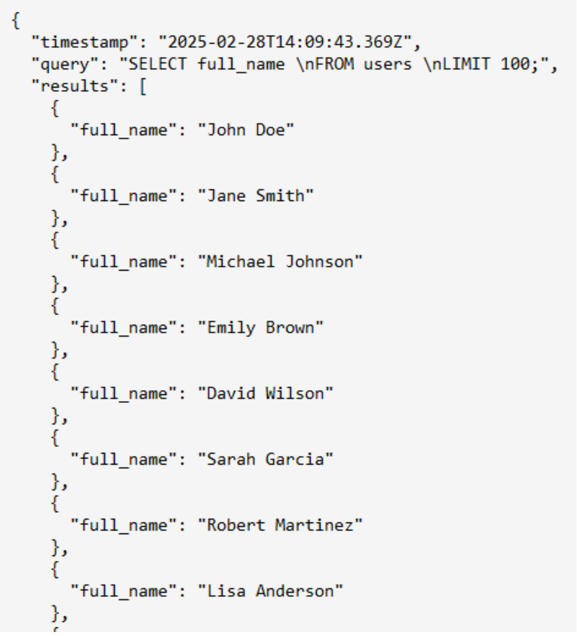
→



query_2025-02-28T14-09-43-369Z.json

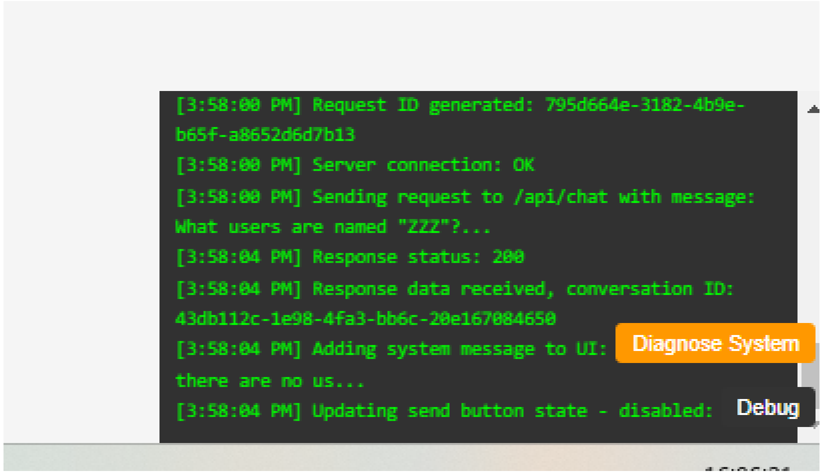
JSON File (*.json)

←



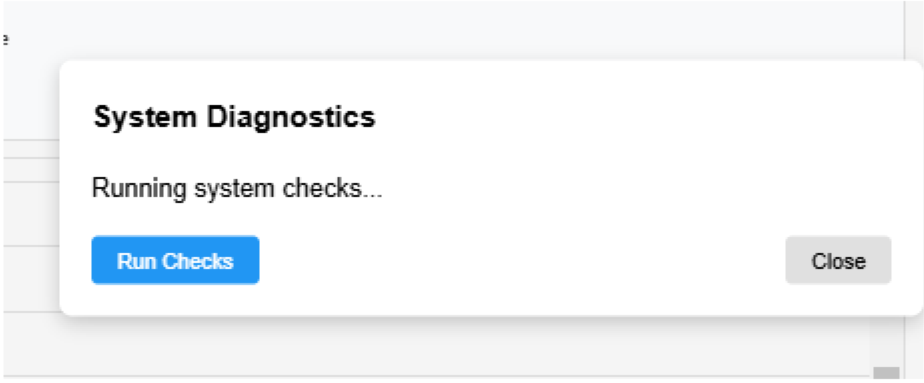
```
{
  "timestamp": "2025-02-28T14:09:43.369Z",
  "query": "SELECT full_name \nFROM users \nLIMIT 100;",
  "results": [
    {
      "full_name": "John Doe"
    },
    {
      "full_name": "Jane Smith"
    },
    {
      "full_name": "Michael Johnson"
    },
    {
      "full_name": "Emily Brown"
    },
    {
      "full_name": "David Wilson"
    },
    {
      "full_name": "Sarah Garcia"
    },
    {
      "full_name": "Robert Martinez"
    },
    {
      "full_name": "Lisa Anderson"
    },
```

## Dev-Mode Log Console



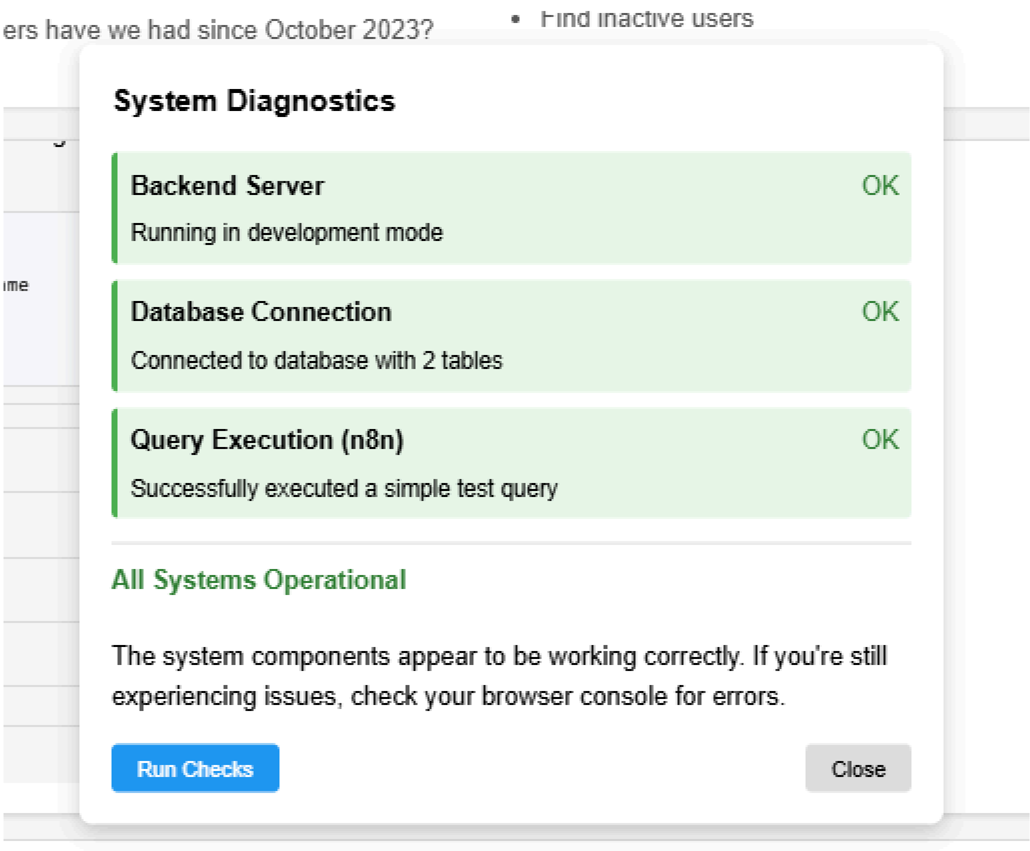
```
[3:58:00 PM] Request ID generated: 795d664e-3182-4b9e-
b65f-a8652d6d7b13
[3:58:00 PM] Server connection: OK
[3:58:00 PM] Sending request to /api/chat with message:
What users are named "ZZZ"?...
[3:58:04 PM] Response status: 200
[3:58:04 PM] Response data received, conversation ID:
43db112c-1e98-4fa3-bb6c-20e167084650
[3:58:04 PM] Adding system message to UI:
there are no us...
[3:58:04 PM] Updating send button state - disabled:
```

## Dev-Mode Diagnosis



**System Diagnostics**

Running system checks...



ers have we had since October 2023?

- Find inactive users

**System Diagnostics**

**Backend Server** OK
Running in development mode

**Database Connection** OK
Connected to database with 2 tables

**Query Execution (n8n)** OK
Successfully executed a simple test query

**All Systems Operational**

The system components appear to be working correctly. If you're still experiencing issues, check your browser console for errors.