# Backend Testing Report

## Objective

The objective of this backend testing was to systematically validate the backend data layer and its exposure through application programming interfaces (APIs). The testing was conducted to ensure that the database schema provided during handover was correctly implemented, structurally sound, and accessible through backend services.

As no executable backend services were delivered as part of the handover, the testing focused on validating database integrity and API-level data accessibility using a controlled local backend environment. The aim was to confirm end-to-end backend readiness for future system integration.

## Testing Scope

The scope of this backend testing included:

- Import and validation of the provided SQL schema in a MySQL database
- Structural verification of all database tables, columns, and relationships
- Insertion of controlled test data into each database table
- Creation of read-only backend APIs to expose database tables
- API-level validation using Postman to verify data retrieval and response integrity

The following database tables were included within scope:

- users
- species_en
- species_tet
- media
- analytics
- changelog

The following aspects were considered out of scope:

- Business logic validation
- Authentication and authorization mechanisms
- Performance, stress, and load testing
- Production deployment and environment-specific testing

## Challenges Encountered

1. Absence of backend APIs in the handover deliverables
2. Incompatibility of PostgreSQL-specific SQL syntax with MySQL
3. Foreign key constraint mismatches during schema import
4. Inability of Postman to interact directly with the database
5. Absence of initial data within database tables

## Mitigation Strategies and Solutions

Each identified challenge was addressed using appropriate mitigation strategies:

### 4.1 Absence of Backend APIs

A lightweight local backend test harness was implemented using Node.js and Express. This backend served exclusively as an intermediary layer to expose database tables via read-only APIs for testing purposes.

### 4.2 SQL Syntax Incompatibility

PostgreSQL-specific constructs, such as the SERIAL data type, were converted to MySQL-compatible definitions (INT AUTO_INCREMENT) to ensure successful schema execution.

### 4.3 Foreign Key Constraint Errors

Data type consistency between primary and foreign keys was reviewed and corrected, ensuring referential integrity across related tables.

### 4.4 Database Access Limitations in Postman

Since Postman operates at the API layer and cannot interact directly with databases, backend APIs were introduced to enable API-level backend testing.

### 4.5 Lack of Test Data

Controlled test records were inserted into each table to facilitate meaningful validation of API responses.

## Backend Testing Methodology

The backend testing was conducted using a structured, stepwise methodology as outlined below.

### Step 1: Database Setup

The SQL schema file was imported into MySQL using MySQL Workbench, followed by confirmation of successful schema creation.

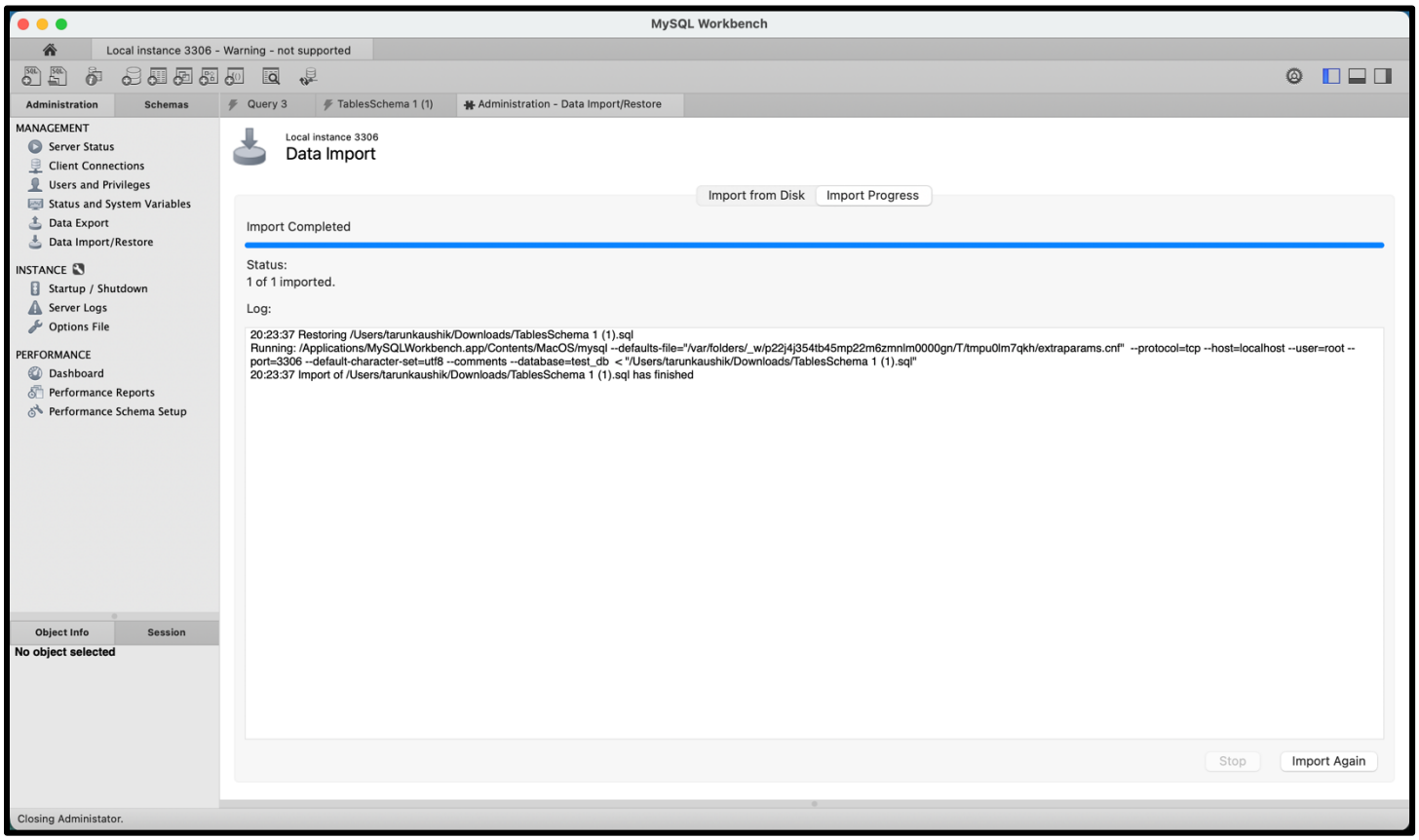


Fig: - Successful import of the backend database schema into the MySQL server using MySQL Workbench's Data Import/Restore feature.

## Step 2: Database Structure Validation

All tables were inspected to verify column definitions, primary keys, foreign keys, and relational integrity.

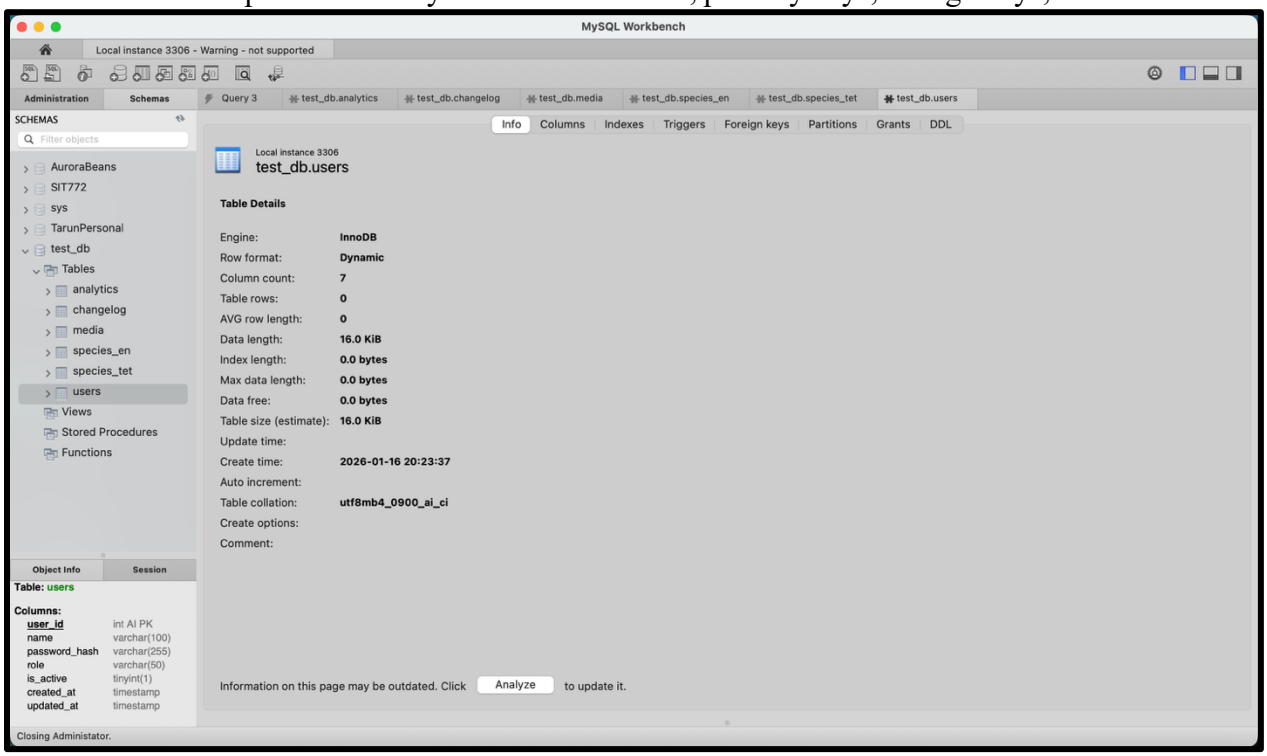


Fig: - Verification of the created database tables (users, species_en, species_tet, media, analytics, changelog) and their column structures in MySQL Workbench.

## Step 3: Test Data Preparation

A single, controlled test record was inserted into each table to support API-level validation.

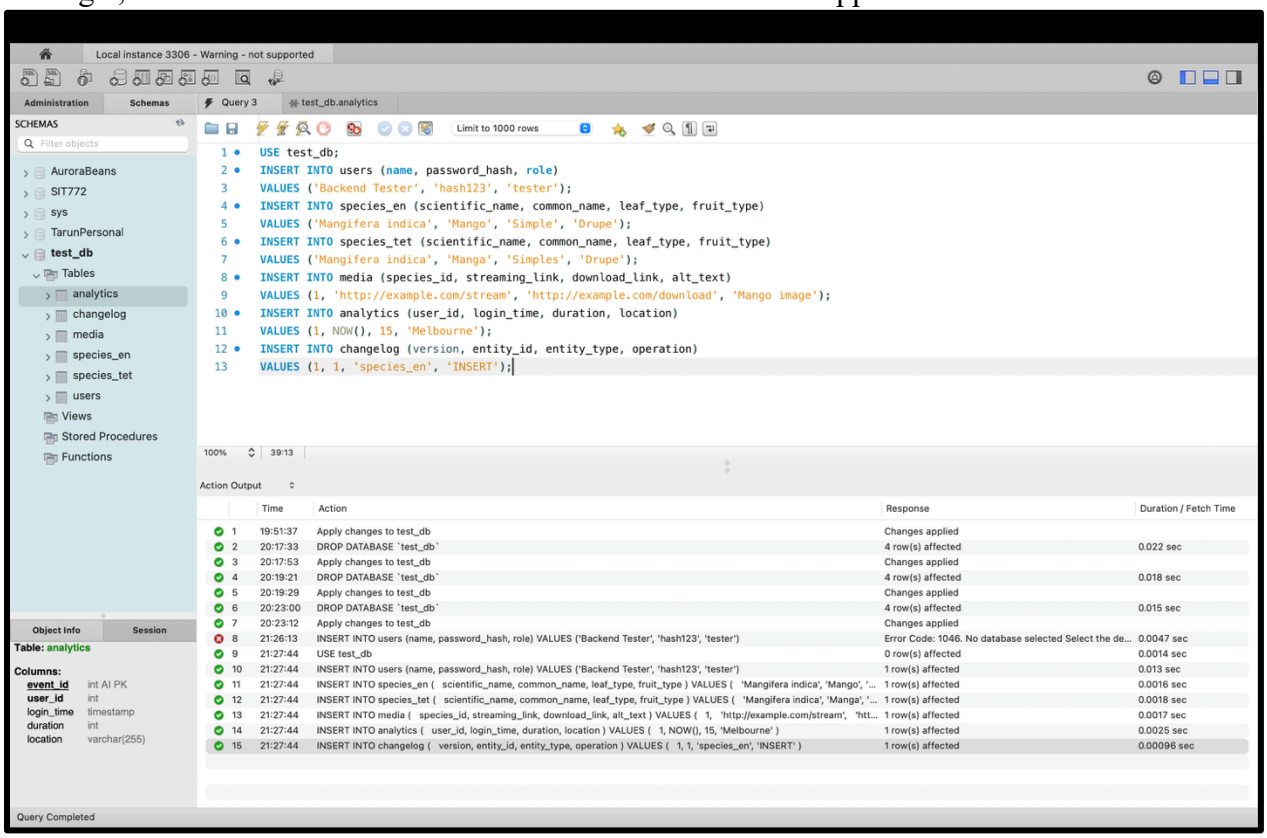


Fig: - Manual insertion of representative test records into all database tables using SQL queries.

Step 4: Backend API Exposure

A local Node.js and Express-based backend was configured to expose one read-only API endpoint per database table.

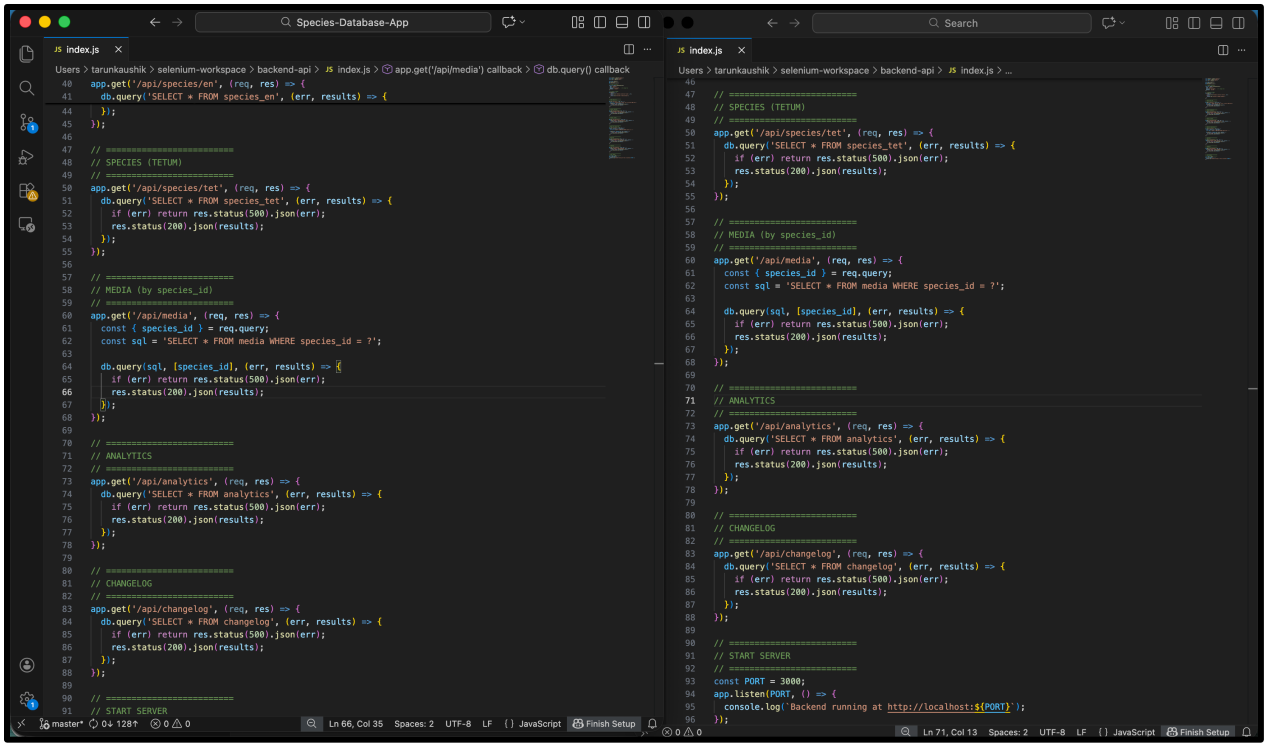


Fig: - Local backend APIs implemented using Node.js and Express to expose database tables for testing.

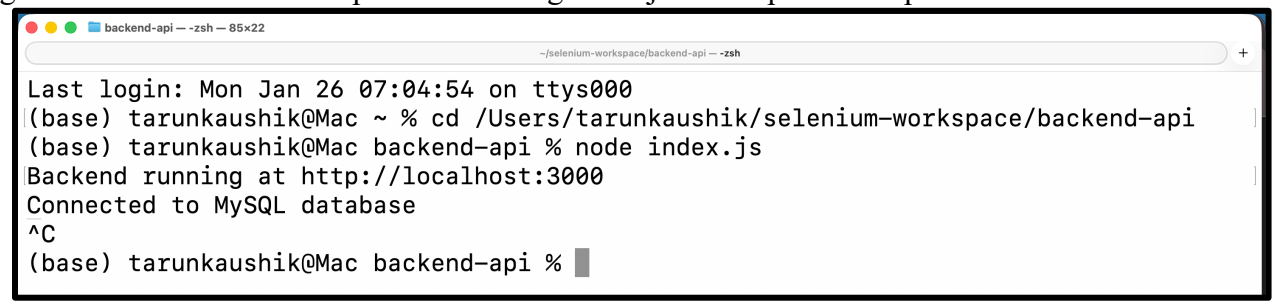


Fig: - Successful startup of local backend server with active database connection.

## Step 5: API Testing Configuration

Postman was configured with environment variables to parameterise the base URL. API requests were organised into table-specific collections for traceability.

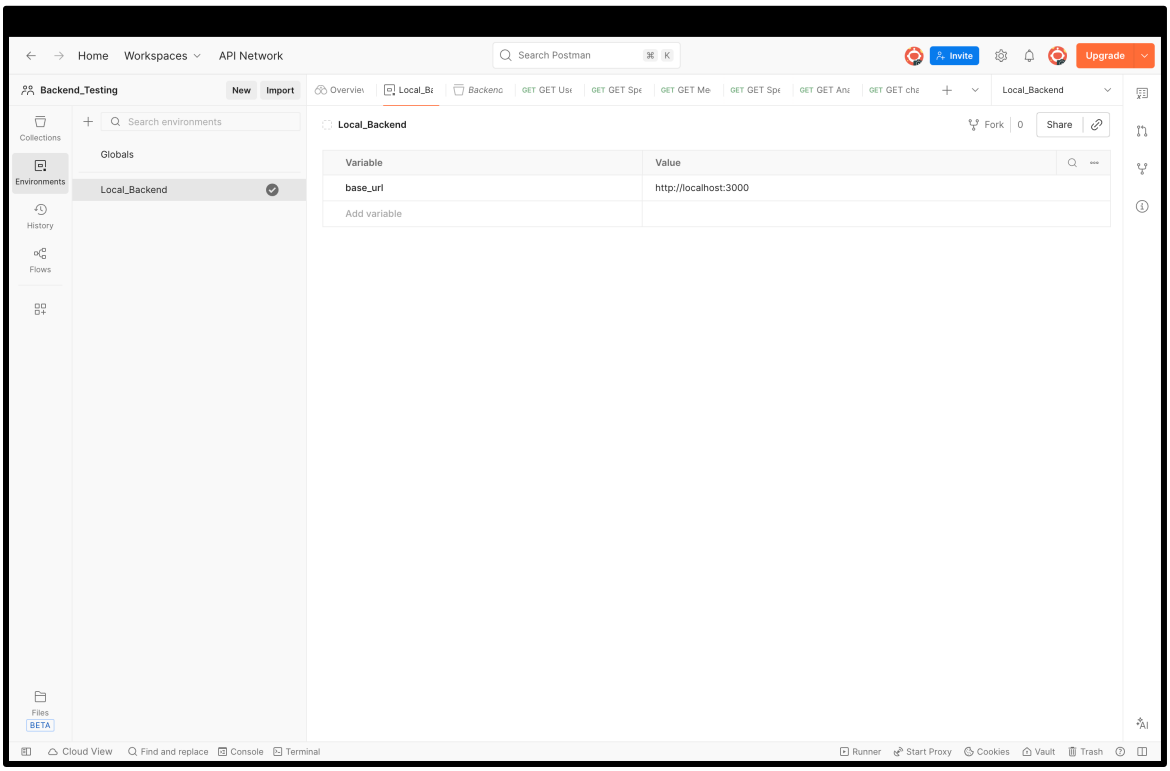


Fig: - Postman environment configuration using parameterised base URL.

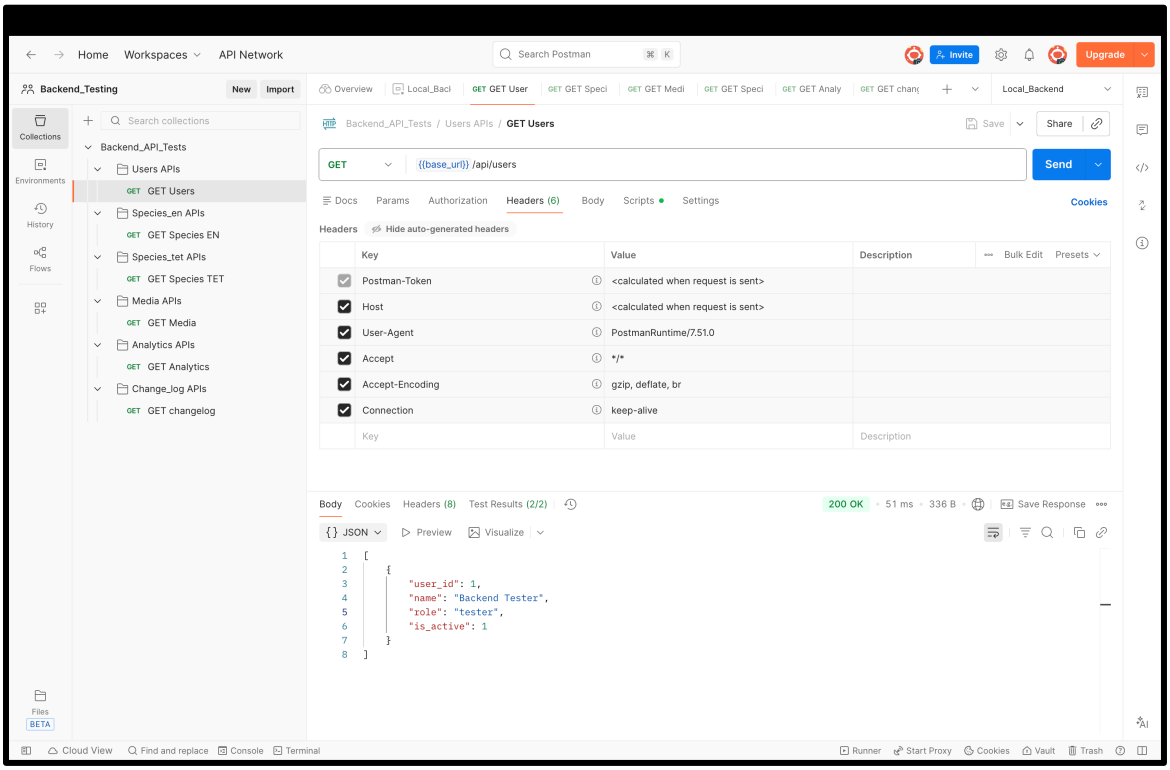


Fig: - Postman collection organised table-wise to maintain traceability between database tables and backend APIs.

## Step 6: API Execution and Validation

Each API endpoint was executed using Postman, validating:

- HTTP status codes
- JSON response structure
- Successful data retrieval from the database

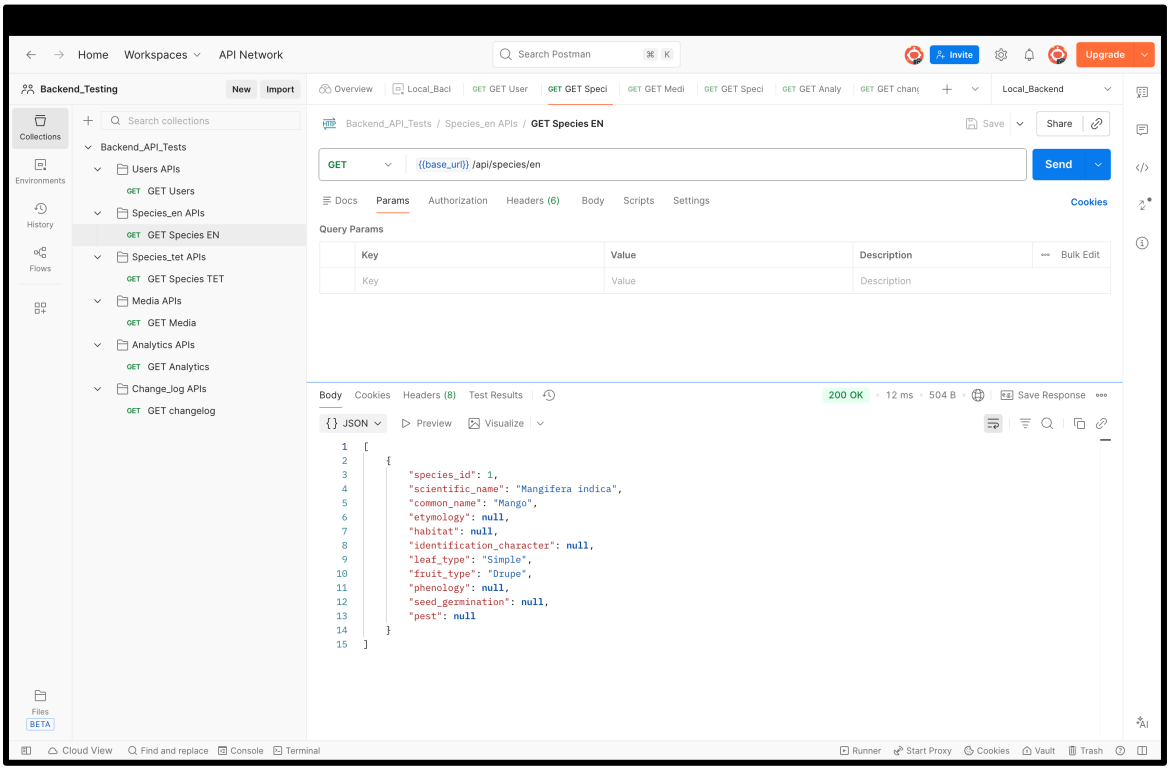


Fig: - Backend API validation for species data using Postman.

## Step 7: Assertion-Based Validation

Standardised Postman test scripts were applied across all APIs to ensure consistent validation criteria.

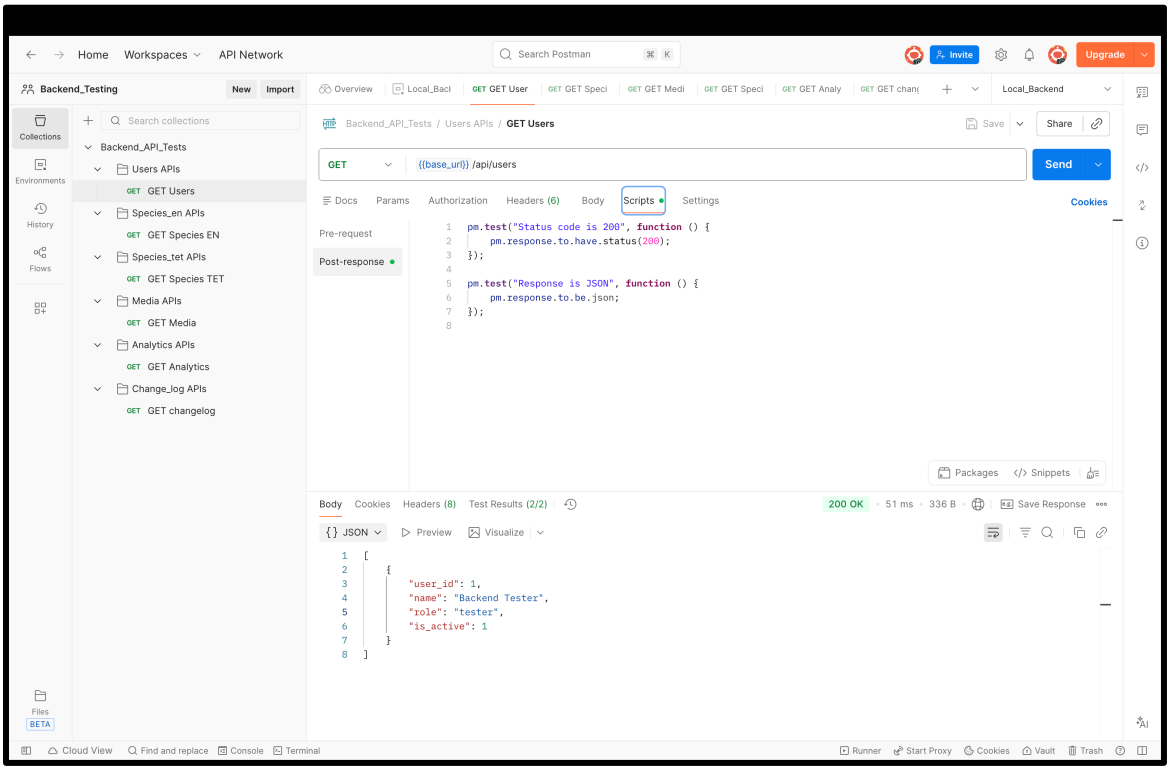


Fig: - Automated backend validation using Postman test scripts.

# Test Results and Observations

The backend testing yielded the following results:

- All APIs returned HTTP 200 (OK) responses
- All responses were returned in valid JSON format
- Data retrieval from MySQL was verified for each table
- No critical or high-severity defects were identified

The backend system demonstrated correct behavior within the defined testing scope.

# Assumptions and Limitations

The following assumptions and limitations applied during testing:

- Backend APIs were created locally solely for testing purposes
- Testing was limited to read-only operations
- No security, performance, or concurrency testing was conducted
- Results are applicable only to the current schema and dataset\

# Future Enhancements and Scope

Future backend testing phases may include:

- Integration testing with frontend applications
- Authentication and authorization testing
- Negative testing and input validation
- Performance and scalability testing
- Security testing (e.g., SQL injection, access control)
- Automation of API tests within CI/CD pipelines

# Conclusion

Backend testing was successfully completed by validating both the database layer and its exposure through backend APIs. The use of a local backend test harness enabled effective API-level testing in the absence of production backend services. All database tables were validated through corresponding APIs, and the backend is considered stable and ready for subsequent integration phases.