Harrier Excel Data Management System

Technical Documentation

Overview

Harrier is a web application designed to streamline the process of uploading and processing Excel data into PostgreSQL databases. It automatically transforms spreadsheet data into structured database tables with appropriate data types, making data integration workflows more efficient.

Architecture

Harrier follows a client-server architecture:

- **Frontend**: React application built with Vite
- **Backend**: Node.js Express server
- **Database**: PostgreSQL
- **File Processing**: Uses xlsx library for parsing Excel files
- **File Upload**: Multer middleware for handling multipart/form-data

Core Features

1. Excel File Processing

Functionality

Converts Excel files (XLSX, XLS, CSV) into PostgreSQL database tables.

Logic Implementation

The `processExcelFile` function in `uploadController.js` handles this process:

1. **File Validation**:
   - Verifies that a file was uploaded
   - Processes the table name from request body, defaulting to "excel_data" if not specified
2. **Table Management**:
   - Checks if the table already exists in the database
   - Handles table conflicts based on the `forceOverwrite` parameter
   - Drops existing tables when overwrite is requested
3. **Excel Parsing**:
   - Reads the uploaded Excel file using the xlsx library
   - Uses the first sheet by default

- Converts sheet data to JSON format

4. **Data Validation**:
   - Finds a valid row with data for schema analysis
   - Returns an error if no valid rows are found
5. **Schema Generation**:
   - Processes column names from the Excel file
   - Sanitizes column names by:
     - Trimming whitespace
     - Replacing spaces with underscores
     - Removing special characters
     - Converting to lowercase
6. **Data Type Detection**:
   - Automatically determines PostgreSQL data types based on the values:
     - Numbers → INTEGER or NUMERIC (based on whether they're integers or decimals)
     - Dates → TIMESTAMP
     - Booleans → BOOLEAN
     - Everything else → TEXT
7. **Table Creation**:
   - Constructs and executes a CREATE TABLE SQL query with appropriate column definitions
8. **Data Insertion**:
   - Maps original column names to sanitized database column names
   - Applies data quality filtering (rejects rows with less than 60% non-empty values)
   - Inserts valid rows into the database
   - Tracks metrics for inserted and rejected rows
9. **Response**:
   - Returns detailed information about the operation:
     - Success status
     - Table name
     - Number of rows inserted
     - Column names
     - Number of rejected rows

Code Example:

```javascript
// Create table with appropriate column types
const columnDefs = sanitizedColumns.map((col, idx) => {
  const val = Object.values(sampleRow)[idx];
  if (typeof val === "number")
    return `${col} ${Number.isInteger(val) ? "INTEGER" : "NUMERIC"}`;
  if (val instanceof Date) return `${col} TIMESTAMP`;
  if (typeof val === "boolean") return `${col} BOOLEAN`;
  return `${col} TEXT`;
});

const createTableQuery = `CREATE TABLE IF NOT EXISTS ${tableName} (${columnDefs.join("
await client.query(createTableQuery);
```

2. Table Existence Checking

Functionality

Allows verification of whether a specific table already exists in the database.

Logic Implementation

The `checkTableExists` function in `uploadController.js` handles this process:

1. **Input Validation**:
   - Verifies that a table name parameter was provided
   - Returns an error if no name is specified
2. **Database Query**:
   - Queries the PostgreSQL information_schema to check for table existence
   - Uses parameterized queries to prevent SQL injection
3. **Response**:
   - Returns JSON with:
     - Success status
     - Boolean indicating if the table exists
     - Table name

Code Example:

```javascript
const result = await client.query(
  "SELECT EXISTS (SELECT FROM information_schema.tables WHERE table_name = $1)",
  [tableName]
);
const exists = result.rows[0].exists;

return res.json({
  success: true,
  exists: exists,
  tableName: tableName,
});
```

3. Data Quality Filtering

Functionality

Automatically filters out rows with insufficient data to maintain database quality.

Logic Implementation

Within the `processExcelFile` function:

1. **Quality Assessment**:
   - For each row, counts the number of non-empty values
   - Calculates the ratio of non-empty values to total columns
   - Rejects rows where this ratio is below 60%
2. **Tracking**:
   - Maintains a count of rejected rows for reporting purposes

Code Example:

```javascript
// Skip rows with less than 60% non-empty values
const nonEmptyValuesCount = Object.values(row).filter(
  (v) => v !== null && v !== ""
).length;

if (nonEmptyValuesCount / originalColumns.length < 0.6) {
  rejectedRowCount++;
  continue; // Skip this row
}
```

4. Table Overwrite Management

Functionality

Provides control over handling existing tables with options to preserve or replace.

Logic Implementation

The `processExcelFile` function implements this with:

1. **Overwrite Parameter**:
   - Reads the `forceOverwrite` parameter from the request body
   - Converts string value to boolean
2. **Conflict Resolution**:
   - If table exists and `forceOverwrite` is false:
     - Returns a 409 Conflict response
     - Informs the user of the table conflict
   - If table exists and `forceOverwrite` is true:
     - Drops the existing table
     - Proceeds with table creation and data insertion

Code Example:

```javascript
// Check if forceOverwrite is set to true
const forceOverwrite = req.body.forceOverwrite === "true";

// Check if the table already exists
const tableExistsResult = await client.query(
  "SELECT EXISTS (SELECT FROM information_schema.tables WHERE table_name = $1)",
  [tableName]
);
const exists = tableExistsResult.rows[0].exists;

if (exists && !forceOverwrite) {
  return res.status(409).json({
    success: false,
    message: `Table '${tableName}' already exists in the database`,
    tableExists: true,
    tableName,
  });
}

// If table exists and forceOverwrite is true, drop the table
if (exists && forceOverwrite) {
  try {
    await client.query(`DROP TABLE IF EXISTS ${tableName}`);
    console.log(`Table '${tableName}' dropped for overwrite`);
  } catch (error) {
    console.error(`Error dropping table ${tableName}:`, error);
    return res.status(500).json({
      success: false,
      message: `Failed to drop existing table '${tableName}'`,
    });
  }
}
```

API Endpoints

Upload Controller Endpoints

1. Check Table Exists

- **Endpoint**: `GET /upload/check-table`
- **Query Parameters**:
  - `name`: The table name to check
- **Response**:

- Success: `{ success: true, exists: boolean, tableName: string }`
- Error: `{ success: false, message: string, exists: false }`

2. Process Excel File

- **Endpoint**: `POST /upload`
- **Request Body**:
  - `file`: The Excel file to upload (multipart/form-data)
  - `tableName`: (Optional) Custom name for the database table
  - `forceOverwrite`: (Optional) Whether to overwrite existing table ("true"/"false")
- **Response**:
  - Success: `{ success: true, message: string, tableName: string, rows: number, columns: string[], rejectedRows: number }`
  - Error: `{ success: false, message: string }`

Security Considerations

1. **SQL Injection Prevention**:
   - Uses parameterized queries for database operations
   - Sanitizes table and column names
2. **Database Connection Management**:
   - Properly manages client connection pool
   - Ensures connections are released in finally blocks
3. **Error Handling**:
   - Robust error handling prevents exposing system details
   - Comprehensive logging for debugging

Configuration Requirements

1. **Database Configuration**:
   - PostgreSQL connection details in `config/database.js`
   - Environment variables for credentials
2. **File Upload Configuration**:
   - Multer middleware setup in `middleware/multerConfig.js`
   - Defines file storage location and naming conventions

Performance Optimizations

1. **Data Filtering**:

- Rejects low-quality rows to maintain database integrity
- Prevents unnecessary storage and processing

2. **Connection Pooling**:
   - Uses PostgreSQL connection pool for efficient resource usage

3. **Error Recovery**:
   - Properly handles errors and releases resources even in failure cases

Best Practices for Usage

1. **File Preparation**:
   - Use consistent column headers
   - Ensure first row contains column names
   - Maintain consistent data types within columns

2. **Table Naming**:
   - Use descriptive table names
   - Avoid special characters and spaces

3. **Data Quality**:
   - Pre-validate Excel data before upload
   - Aim for complete data rows to avoid filtering