Part 6: Upload Routes

JSON Conversion Route

```
javascript

app.post('/upload/json', upload.single('file'), (req, res) => {
  try {
    let workbook = XLSX.read(req.file# Complete server.js Explanation

## Part 1: Importing Dependencies (The Tools You Need)

```javascript
const fs = require('fs');
const XLSX = require('xlsx');
const express = require('express')
const app = express();
const port = 3000;
const multer = require('multer');
const PDFDocument = require('pdfkit');
const sqlite3 = require('sqlite3').verbose();
const session = require('express-session');
const bcrypt = require('bcrypt');
```

What each tool does (SUPER DETAILED):

`fs` (File System):

- Built into Node.js (no need to install)
- Functions like `fs.readFile()`, `fs.writeFile()`, `fs.unlink()`
- In your code: Not directly used but imported (maybe for future use)
- Example: `fs.readFile('data.txt', 'utf8', callback)` reads a text file
- Works with both synchronous (blocking) and asynchronous (non-blocking) operations

`XLSX` (Excel Library):

- External library (needs `npm install xlsx`)
- Can read: .xlsx, .xls, .csv, .ods files
- Can write: Excel files in multiple formats
- Main functions: `XLSX.read()` (read files), `XLSX.utils.sheet_to_json()` (convert to JavaScript)
```

- Supports formulas, formatting, multiple sheets, charts
- Memory efficient - processes large files without loading everything at once

`express` (Web Framework):

- External library (needs `npm install express`)
- Built on top of Node.js's built-in `http` module
- Handles: routing, middleware, request/response objects
- `const app = express()` creates your web application instance
- Provides methods like `app.get()`, `app.post()`, `app.use()`
- Middleware system allows plugging in functionality (like authentication, logging)

`port = 3000`:

- Port number where your server listens for connections
- Ports 0-1023 are reserved for system services
- Port 3000 is commonly used for development
- Users access your app at `http://localhost:3000`
- In production, you'd typically use port 80 (HTTP) or 443 (HTTPS)

`multer` (File Upload Handler):

- External library (needs `npm install multer`)
- Built specifically for handling `multipart/form-data` (file uploads)
- Without multer, Express can't handle file uploads
- Provides different storage options: memory, disk, custom
- Can limit file size, filter file types, rename files
- Processes files and makes them available in `req.file` or `req.files`

`PDFDocument` (PDF Creator):

- External library (needs `npm install pdfkit`)
- Creates PDF files programmatically
- Can add: text, images, shapes, tables, fonts
- Supports: multiple pages, styling, positioning
- Streams data (memory efficient for large PDFs)

- Alternative to libraries like jsPDF or Puppeteer

`sqlite3` **(Database):**

- External library (needs `npm install sqlite3`)
- `.verbose()` enables detailed error messages and debugging
- SQLite is file-based database (no separate server needed)
- Supports SQL queries, transactions, foreign keys
- Database file created automatically if it doesn't exist
- Alternative to MySQL, PostgreSQL for simple applications

`session` **(Session Management):**

- External library (needs `npm install express-session`)
- Stores user data between HTTP requests
- By default, stores in memory (lost when server restarts)
- Can be configured to store in database, Redis, files
- Creates session ID cookie in user's browser
- Essential for login systems (remembers who's logged in)

`bcrypt` **(Password Security):**

- External library (needs `npm install bcrypt`)
- Uses bcrypt hashing algorithm (designed for passwords)
- Much slower than regular hashing (this is good for security)
- Includes salt generation (prevents rainbow table attacks)
- One-way function: can hash password but can't unhash
- `bcrypt.hash()` creates hash, `bcrypt.compare()` verifies password

Part 2: File Upload Configuration

```javascript
const upload = multer({
  storage: multer.memoryStorage(),
  fileFilter: (req, file, cb) => {
    if (file.mimetype.includes('spreadsheet')) cb(null, true);
    else cb(new Error('Invalid file type'), false);
  }
});
```

SUPER DETAILED Breakdown:

`const upload = multer({...})`:

- Creates a multer instance with specific configuration
- This `upload` variable becomes middleware you can use in routes
- Configuration object defines how files are handled

`storage: multer.memoryStorage()`:

- **Memory Storage:** Files stored in RAM as Buffer objects
- **Alternative options:**
  - `multer.diskStorage()` - saves files to disk
  - Custom storage engines
- **Memory storage pros:** Fast access, no disk cleanup needed
- **Memory storage cons:** Uses RAM, files lost if server crashes
- **Buffer object:** Raw binary data in memory (like `req.file.buffer`)
- **When to use:** Small files, temporary processing, don't need permanent storage

`fileFilter: (req, file, cb) => {...}`:

- **Purpose:** Decides which files to accept/reject BEFORE processing
- **Parameters:**
  - `req` = HTTP request object (contains user info, session, etc.)
  - `file` = File information object with properties:
    - `file.fieldname` = name of form field
    - `file.originalname` = original filename on user's computer
    - `file.encoding` = file encoding type

    - `file.mimetype` = MIME type (like 'application/vnd.openxmlformats-officedocument.spreadsheetml.sheet')
    - `file.size` = file size in bytes
  - `cb` = callback function to call when decision is made

`if (file.mimetype.includes('spreadsheet'))`:

- **MIME types for Excel files:**
  - `.xlsx` = 'application/vnd.openxmlformats-officedocument.spreadsheetml.sheet'
  - `.xls` = 'application/vnd.ms-excel'
  - `.csv` = 'text/csv'
- **`.includes('spreadsheet')`:** Checks if the word "spreadsheet" appears anywhere in the MIME type
- **Why this works:** Excel MIME types contain "spreadsheet" in their names
- **Alternative approach:** Could check exact MIME types or file extensions

`cb(null, true)` vs `cb(new Error('Invalid file type'), false)`:

- **Callback pattern:** First parameter is error (null if no error), second is result
- **`cb(null, true)`:**
  - No error occurred
  - Accept this file
  - File will be processed and available in `req.file`
- **`cb(new Error('Invalid file type'), false)`:**
  - An error occurred
  - Reject this file
  - Error message will be sent to client
  - Processing stops here for this file

How this works in practice:

1. User uploads file through HTML form
2. Multer intercepts the upload
3. `fileFilter` function runs BEFORE file is stored
4. If file is accepted, it's stored in memory as Buffer
5. If file is rejected, error is returned to user

6. Accepted files become available in route handlers as `req.file`

Part 3: Database Setup

```javascript
const db = new sqlite3.Database('./excel_data.db');

db.run(`CREATE TABLE IF NOT EXISTS users (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL,
  email TEXT NOT NULL UNIQUE,
  passwordHash TEXT NOT NULL
)`);

db.run(`CREATE TABLE IF NOT EXISTS settings (
  userId INTEGER PRIMARY KEY,
  storeExcelConversions INTEGER DEFAULT 0,
  FOREIGN KEY(userId) REFERENCES users(id)
)`);
```

SUPER DETAILED Database Breakdown:

`const db = new sqlite3.Database('./excel_data.db')`:

- **File location:** `./excel_data.db` means current directory
- **Auto-creation:** If file doesn't exist, SQLite creates it automatically
- **File format:** Binary SQLite database file (not human-readable)
- **Connection:** This creates persistent connection to database
- **Threading:** SQLite handles concurrent access automatically
- **Alternative constructors:**
  - `new sqlite3.Database(':memory:')` = in-memory database (lost when app stops)
  - `new sqlite3.Database('path/to/file.db')` = custom file location

`db.run()` Method:

- **Purpose:** Execute SQL statements that don't return data (CREATE, INSERT, UPDATE, DELETE)
- **Asynchronous:** Doesn't block other code while running
- **Callback:** Optional third parameter for handling completion/errors
- **Return value:** Returns Database object for chaining

- **Alternative methods:**
  - `db.get()` = get single row
  - `db.all()` = get multiple rows
  - `db.each()` = process rows one by one

Users Table Deep Dive:

`CREATE TABLE IF NOT EXISTS users`:

- **IF NOT EXISTS:** Only create if table doesn't already exist (prevents errors)
- **Why important:** App might restart multiple times, this prevents duplicate table errors
- **Alternative:** `CREATE TABLE users` would fail if table exists

Column Analysis:

`id INTEGER PRIMARY KEY AUTOINCREMENT`:

- **INTEGER type:** Whole numbers only (-2,147,483,648 to 2,147,483,647)
- **PRIMARY KEY:** Unique identifier for each row, indexed automatically
- **AUTOINCREMENT:** SQLite automatically assigns next available number
- **Why important:** Every table needs unique way to identify rows
- **Index creation:** SQLite automatically creates index on primary key (fast lookups)
- **Sequence:** 1, 2, 3, 4... even if you delete rows, numbers don't reuse

`name TEXT NOT NULL`:

- **TEXT type:** Variable-length string, up to 1 billion characters
- **NOT NULL constraint:** Field cannot be empty, must have value
- **Storage:** UTF-8 encoding (supports international characters)
- **Comparison:** Case-sensitive by default
- **Alternative types:** VARCHAR(50) works but TEXT is more flexible

`email TEXT NOT NULL UNIQUE`:

- **UNIQUE constraint:** No two users can have same email address
- **Practical effect:** Prevents duplicate accounts
- **Index creation:** SQLite automatically creates index on UNIQUE columns
- **Case sensitivity:** 'User@Email.com' different from 'user@email.com'

- **Validation:** Database enforces uniqueness, app should also validate email format

`passwordHash TEXT NOT NULL`**:**

- **Purpose:** Stores bcrypt-hashed version of password
- **Never plain text:** Original password never stored
- **Hash length:** bcrypt hashes are typically 60 characters
- **Security:** Even if database is stolen, passwords are protected
- **One-way:** Cannot convert hash back to original password

Settings Table Deep Dive:

`userId INTEGER PRIMARY KEY`**:**

- **No AUTOINCREMENT:** Each user has exactly one settings record
- **Primary key:** Uses same ID as user's ID
- **One-to-one relationship:** Each user ID appears maximum once in settings table

`storeExcelConversions INTEGER DEFAULT 0`**:**

- **INTEGER for boolean:** SQLite doesn't have true boolean type
- **0 = false, 1 = true:** Standard convention for boolean values
- **DEFAULT 0:** If not specified, defaults to false (don't store conversions)
- **Purpose:** User preference for whether to save their Excel conversion history

`FOREIGN KEY(userId) REFERENCES users(id)`**:**

- **Relationship enforcement:** userId must exist in users table
- **Referential integrity:** Can't create settings for non-existent user
- **Cascade options:** Could add ON DELETE CASCADE to auto-delete settings when user deleted
- **Index:** SQLite may create index on foreign key columns
- **Performance:** Helps optimize joins between tables

Database Schema Visualization:

```
users table:
┌────┬──────────┬──────────────────────┬──────────────────────────────┐
│ id │  name    │        email         │        passwordHash          │
├────┼──────────┼──────────────────────┼──────────────────────────────┤
│ 1  │ John     │ john@example.com     │ $2b$10$abcd...               │
│ 2  │ Jane     │ jane@example.com     │ $2b$10$efgh...               │
└────┴──────────┴──────────────────────┴──────────────────────────────┘

settings table:
┌────────┬────────────────────────┐
│ userId │ storeExcelConversions  │
├────────┼────────────────────────┤
│ 1      │ 1 (true)               │
│ 2      │ 0 (false)              │
└────────┴────────────────────────┘
```

SQL Execution Flow:

1. App starts up
2. SQLite opens/creates excel_data.db file
3. First `db.run()` executes: creates users table if needed
4. Second `db.run()` executes: creates settings table if needed
5. Database is ready for use
6. Tables persist even after app shuts down

Part 4: Static Files

```javascript
app.use(express.static('public'));
```

SUPER DETAILED Static File Serving:

`app.use()` **Method:**

- **Purpose:** Mount middleware that runs for ALL requests
- **Order matters:** Middleware runs in the order it's defined
- **Global scope:** This affects every HTTP request to your server
- **Middleware function:** `express.static()` is built-in Express middleware

`express.static('public')` **Deep Dive:**

What it does:

- **File server:** Turns your Express app into a file server for static assets
- **Directory mapping:** Maps URLs to files in the 'public' directory
- **Automatic MIME types:** Express automatically sets correct Content-Type headers
- **Caching:** Sends appropriate cache headers for browser optimization

URL to File Mapping Examples:

```
Request URL              →  File Path
/style.css               →  public/style.css
/js/app.js               →  public/js/app.js
/images/logo.png         →  public/images/logo.png
/favicon.ico             →  public/favicon.ico
/                        →  public/index.html (if exists)
```

File Types Typically Served:

- **CSS files:** Stylesheets for webpage appearance
- **JavaScript files:** Client-side scripts that run in browser
- **Images:** PNG, JPG, GIF, SVG, etc.
- **Fonts:** WOFF, TTF, EOT files
- **HTML files:** Static pages (though your app generates dynamic HTML too)
- **Documents:** PDFs, text files for download

Behind the Scenes Process:

1. **Request comes in:** User requests `/style.css`
2. **Middleware check:** Express.static middleware runs first
3. **File lookup:** Checks if `public/style.css` exists
4. **File found:** Reads file from disk
5. **Headers set:** Sets Content-Type to `text/css`
6. **Response sent:** Streams file content to browser
7. **File not found:** Continues to next middleware/route

Security Features:

- **Path traversal protection:** Can't use `../` to access files outside public directory
- **Hidden file protection:** Files starting with `.` are not served by default
- **Directory listing:** Doesn't show directory contents if no index file

Performance Optimizations:

- **ETag headers:** Browser can check if file changed before downloading
- **Last-Modified headers:** Browser caching based on file modification time
- **Conditional requests:** Returns 304 Not Modified if file unchanged
- **Static file caching:** Express caches file stats for better performance

Configuration Options (if you wanted to customize):

```javascript
app.use(express.static('public', {
  maxAge: '1d',        // Cache files for 1 day
  index: false,        // Don't serve index.html automatically
  dotfiles: 'deny',    // Deny access to hidden files
  extensions: ['html'] // Auto-append .html to requests
}));
```

Directory Structure Example:

```
project/
├── server.js
├── public/           ← This is what express.static serves
│   ├── style.css     ← Accessible at /style.css
│   ├── app.js        ← Accessible at /app.js
│   ├── images/
│   │   └── logo.png  ← Accessible at /images/logo.png
│   └── index.html    ← Accessible at /
└── node_modules/
```

Why Static Files are Separate:

- **Client-side code:** Runs in user's browser, not on server
- **No processing needed:** Files sent exactly as they are
- **Faster serving:** No need to generate content dynamically
- **CDN friendly:** Can easily move to Content Delivery Network later

Part 5: Data Cleaning Functions

```javascript
function cleanExcelData(data) {
  return data
    .filter(row => {
      // Remove empty rows
      return Object.values(row).some(val => val !== "");
    })
    .map(row => {
      const cleanRow = {};
      for (const [key, value] of Object.entries(row)) {
        // Clean keys and values
        const cleanKey = key.toString().trim();

        // Clean different value types
        let cleanValue;
        if (typeof value === 'string') {
          cleanValue = value.trim().replace(/\s+/g, ' ');
        } else if (value instanceof Date) {
          cleanValue = value.toISOString().split('T')[0]; // Format dates
        } else {
          cleanValue = value;
        }

        cleanRow[cleanKey] = cleanValue;
      }
      return cleanRow;
    });
}
```

SUPER DETAILED Data Cleaning Breakdown:

Function Purpose:

- **Input:** Raw JSON data from Excel file (often messy)
- **Output:** Clean, standardized data ready for processing
- **Why needed:** Excel files often contain empty rows, extra spaces, inconsistent formatting

`return data.filter(...).map(...)`:

- **Method chaining:** Combines filter and map operations

- **Functional programming:** No side effects, returns new array
- **Order matters:** Filter first (remove bad rows), then map (clean good rows)

Step 1: `filter()` Operation

`data.filter(row => {...})`:

- **Purpose:** Remove completely empty rows from dataset
- **Input:** Each `row` is an object like `{Name: "John", Age: 30, City: ""}`
- **Returns:** New array with only rows that pass the test

`Object.values(row)`:

- **What it does:** Gets all values from object, ignoring keys
- **Example:** `{Name: "John", Age: 30}` becomes `["John", 30]`
- **Why useful:** We want to check if ANY value in row has data

`Object.values(row).some(val => val !== "")`:

- **`.some()` method:** Returns true if ANY value passes the test
- **Test condition:** `val !== ""` checks if value is not empty string
- **Logic:** "Keep this row if at least one cell has data"
- **Edge cases handled:**
  - `null` values pass test (not equal to "")
  - `0` values pass test (not equal to "")
  - `false` values pass test (not equal to "")
  - Only empty strings `""` fail the test

Real Examples:

```javascript
// This row would be KEPT (has data):
{Name: "John", Age: "", City: "NYC"} → some() returns true (Name has data)

// This row would be REMOVED (no data):
{Name: "", Age: "", City: ""} → some() returns false (all empty strings)

// This row would be KEPT (zero is data):
{Name: "", Age: 0, City: ""} → some() returns true (Age has value)
```

Step 2: `map()` Operation

`data.map(row => {...})`:

- **Purpose:** Transform each row, cleaning keys and values
- **Input:** Each `row` that passed the filter
- **Returns:** New array with cleaned row objects

`const cleanRow = {}`:

- **Fresh object:** Start with empty object for each row
- **Avoid mutation:** Don't modify original row object
- **Clean slate:** Ensures no leftover properties

`for (const [key, value] of Object.entries(row))`:

- **Object.entries():** Converts object to array of [key, value] pairs
- **Example:** `{Name: "John", Age: 30}` becomes `[["Name", "John"], ["Age", 30]]`
- **Destructuring:** `[key, value]` extracts both parts at once
- **Iteration:** Processes each property of the row object

Key Cleaning: **`const cleanKey = key.toString().trim()`:**

- **`.toString()`:** Converts key to string (handles edge cases where key might not be string)
- **`.trim()`:** Removes whitespace from beginning and end
- **Why needed:** Excel column headers often have extra spaces
- **Example:** `" Name "` becomes `"Name"`

Value Cleaning Logic:

String Values:

```javascript
if (typeof value === 'string') {
  cleanValue = value.trim().replace(/\s+/g, ' ');
}
```

- `typeof value === 'string'`: Check if value is text
- `.trim()`: Remove leading/trailing whitespace
- `.replace(/\s+/g, ' ')`: Replace multiple spaces with single space
  - `/\s+/g` is regular expression:
    - `\s` = any whitespace character (space, tab, newline)
    - `+` = one or more consecutive whitespace chars
    - `g` = global flag (replace all occurrences, not just first)
- **Example:** `"  John    Doe  "` becomes `"John Doe"`

Date Values:

```javascript
else if (value instanceof Date) {
  cleanValue = value.toISOString().split('T')[0];
}
```

- `instanceof Date`: Check if value is Date object
- `.toISOString()`: Convert to standard format: `"2024-03-15T14:30:00.000Z"`
- `.split('T')[0]`: Split on 'T' and take first part (date only)
- **Result:** `"2024-03-15"` (YYYY-MM-DD format)
- **Why needed:** Excel dates can be in various formats, this standardizes them

Other Values:

```javascript
else {
  cleanValue = value;
}
```

- **No processing:** Numbers, booleans, null values kept as-is
- **Handles:** Integers, floats, true/false, null, undefined

Final Assignment:

```javascript
cleanRow[cleanKey] = cleanValue;
```

- **Property assignment:** Add cleaned key-value pair to new object
- **Result:** Fresh object with cleaned keys and values

Complete Example:

```javascript
// Input row (messy):
{
  "  Name  ": "  John    Doe  ",
  " Age ": 30,
  "Birth Date": new Date("2024-03-15T14:30:00.000Z"),
  "City": "",
  " Country ": "   USA   "
}

// After cleaning:
{
  "Name": "John Doe",
  "Age": 30,
  "Birth Date": "2024-03-15",
  "City": "",
  "Country": "USA"
}
```

Validation Function:

```javascript
function validateData(data) {
  const errors = [];
  data.forEach((row, i) => {
    // Add validation logic here
    if (!row.Name) errors.push({row: i+1, message: "Missing Name"});
  });
  return errors;
}
```

DETAILED Validation Breakdown:

`const errors = []`:

- **Error collection:** Array to store all validation problems found
- **Structure:** Each error is object with row number and message

`data.forEach((row, i) => {...})`:

- **Iteration:** Process each row in cleaned data
- **Parameters:**
  - `row` = current row object
  - `i` = zero-based index (0, 1, 2, ...)
- **Purpose:** Check each row for data quality issues

`if (!row.Name)`:

- **Falsy check:** Tests if Name property is falsy
- **Falsy values:** `undefined`, `null`, `""`, `0`, `false`
- **Logic:** "If Name field is missing or empty, it's an error"
- **Expandable:** Could add more validation rules here

`errors.push({row: i+1, message: "Missing Name"})`:

- **Error object:** Structured information about the problem
- `row: i+1`: Convert zero-based index to human-readable row number
- `message`: Human-readable description of the problem
- **Result:** Errors array contains all validation issues found

Example validation output:

```javascript
[
  {row: 3, message: "Missing Name"},
  {row: 7, message: "Missing Name"},
  {row: 12, message: "Missing Name"}
]
```

Why These Functions Matter:

- **Data quality:** Garbage in, garbage out - clean data produces better results
- **User experience:** Clear error messages help users fix their data
- **Reliability:** Prevents crashes from malformed data
- **Consistency:** Standardized format makes processing predictable

Part 6: Upload Routes

JSON Conversion Route

```javascript
app.post('/upload/json', upload.single('file'), (req, res) => {
  try {
    let workbook = XLSX.read(req.file.buffer, { type: 'buffer' });
    let sheetName = workbook.SheetNames[0];
    let jsonData = XLSX.utils.sheet_to_json(workbook.Sheets[sheetName]);
    res.json(jsonData);
  } catch(e) {
    res.send('invalid'+e)
  }
})
```

What happens:

1. User uploads Excel file
2. `XLSX.read()` = Convert Excel file to workbook object
3. `workbook.SheetNames[0]` = Get name of first sheet
4. `sheet_to_json()` = Convert sheet data to JavaScript array
5. `res.json()` = Send data back as JSON

PDF Conversion Route

```
javascript
```

```
app.post('/upload/pdf', upload.single('file'), (req, res) => {
  if (!req.file) {
    return res.status(400).send('No file uploaded');
  }
  res.setHeader('Content-Type', 'application/pdf');
  res.setHeader('Content-Disposition', 'attachment; filename="report.pdf"');

  let workbook = XLSX.read(req.file.buffer, { type: 'buffer' });
  let sheetName = workbook.SheetNames[0];
  let jsonData = XLSX.utils.sheet_to_json(workbook.Sheets[sheetName]);
  let cleanData = cleanExcelData(jsonData);
  let errors = validateData(cleanData);

  const doc = new PDFDocument();
  doc.pipe(res);

  doc.fontSize(20).text('Excel to PDF Report', { align: 'center' });

  // Add errors section
  doc.fontSize(16).text('Data Issues:', { underline: true });
  if (errors.length === 0) {
    doc.text('No issues found.');
  } else {
    errors.forEach(err => {
      doc.text(`• Row ${err.row}: ${err.message}`);
    });
  }

  // Add data table
  doc.addPage();
  doc.fontSize(16).text('Processed Data', { align: 'center' });

  // Print table headers
  if (cleanData.length > 0) {
    const columns = Object.keys(cleanData[0]);
    doc.moveDown(0.5);
    doc.fontSize(12).font('Helvetica-Bold').text(columns.join(' | '));
    doc.moveDown(0.2);
    doc.font('Helvetica');
    cleanData.forEach((row, i) => {
      if (i < 2000000) {
        const rowText = columns.map(col => row[col] !== undefined ? String(row[col]) : '').joir
        doc.text(rowText);
```

```
      }
    });
  } else {
    doc.moveDown().text('No data found.');
  }

  doc.end();
})
```

PDF Creation Process:

1. Set headers to tell browser this is a PDF download
2. Convert Excel to JSON (same as before)
3. Clean and validate the data
4. Create new PDF document
5. Add title "Excel to PDF Report"
6. List any data problems found
7. Add new page
8. Create table with column headers
9. Add each row of data
10. Send PDF to user

SQL Export Route

```javascript
app.post('/upload/sql', upload.single('file'), (req, res) => {
  if (!req.file) {
    return res.status(400).send('No file uploaded');
  }
  let workbook = XLSX.read(req.file.buffer, { type: 'buffer' });
  let sheetName = workbook.SheetNames[0];
  let jsonData = XLSX.utils.sheet_to_json(workbook.Sheets[sheetName]);
  let cleanData = cleanExcelData(jsonData);

  if (!cleanData.length) {
    return res.status(400).send('No data found in Excel file.');
  }

  // Get columns from the first row
  const columns = Object.keys(cleanData[0]);
  const tableName = 'excel_data'; // You can change this

  // Generate CREATE TABLE statement
  let sql = `CREATE TABLE IF NOT EXISTS ${tableName} (\n  id INTEGER PRIMARY KEY AUTOINCREMENT,
  sql += columns.map(col => `  [${col}] TEXT`).join(',\n') + '\n);\n\n';

  // Generate INSERT statements
  cleanData.forEach(row => {
    const values = columns.map(col => {
      const val = row[col] == null ? '' : row[col].toString().replace(/'/g, "''");
      return `'${val}'`;
    });
    sql += `INSERT INTO ${tableName} (${columns.map(col => `[${col}]`).join(', ')}) VALUES (${v
  });

  res.setHeader('Content-Type', 'application/sql');
  res.setHeader('Content-Disposition', 'attachment; filename="excel_data.sql"');
  res.send(sql);
});
```

SQL Generation Process:

1. Convert Excel to clean JSON data
2. Get column names from first row
3. Create `CREATE TABLE` statement with all columns

4. For each row, create an `INSERT` statement
5. Escape single quotes in data (`'` becomes `''`)
6. Send as downloadable .sql file

Part 7: Session Management

```javascript
app.use(session({
  secret: 'your-secret-key', // Change this to a strong secret in production
  resave: false,
  saveUninitialized: false,
  cookie: { secure: false } // Set to true if using HTTPS
}));

app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

Session Configuration:

- `secret` = Key used to encrypt session data (should be random and secret)
- `resave: false` = Don't save session if nothing changed
- `saveUninitialized: false` = Don't save empty sessions
- `cookie: { secure: false }` = Allow cookies over HTTP (for development)

Body Parsers:

- `express.json()` = Parse JSON data from requests
- `express.urlencoded()` = Parse form data from requests

Part 8: User Authentication

Registration

```javascript
app.post('/register', async (req, res) => {
  const { name, email, password } = req.body;
  if (!name || !email || !password) {
    return res.status(400).json({ error: 'All fields required.' });
  }
  if (password.length < 8) {
    return res.status(400).json({ error: 'Password must be at least 8 characters.' });
  }
  db.get('SELECT * FROM users WHERE email = ?', [email], async (err, user) => {
    if (err) return res.status(500).json({ error: 'Database error.' });
    if (user) return res.status(400).json({ error: 'Email already registered.' });
    const passwordHash = await bcrypt.hash(password, 10);
    db.run('INSERT INTO users (name, email, passwordHash) VALUES (?, ?, ?)', [name, email, pass
      if (err) return res.status(500).json({ error: 'Registration failed.' });
      res.json({ success: true, message: 'Registration successful.' });
    });
  });
});
```

Registration Process:

1. Extract name, email, password from request
2. Validate all fields are present
3. Check password is at least 8 characters
4. Check if email is already used
5. Hash password with bcrypt (makes it unreadable)
6. Save new user to database
7. Send success response

Login

```javascript
app.post('/login', (req, res) => {
  const { email, password } = req.body;
  if (!email || !password) {
    return res.status(400).json({ error: 'All fields required.' });
  }
  db.get('SELECT * FROM users WHERE email = ?', [email], async (err, user) => {
    if (err) return res.status(500).json({ error: 'Database error.' });
    if (!user) return res.status(400).json({ error: 'Invalid credentials.' });
    const match = await bcrypt.compare(password, user.passwordHash);
    if (!match) return res.status(400).json({ error: 'Invalid credentials.' });
    req.session.userId = user.id;
    res.json({ success: true, message: 'Login successful.', name: user.name, email: user.email
  });
});
```

Login Process:

1. Get email and password from request
2. Find user in database by email
3. Compare provided password with stored hash
4. If passwords match, save user ID in session
5. Send success response with user info

Logout

```javascript
app.post('/logout', (req, res) => {
  req.session.destroy(() => {
    res.json({ success: true, message: 'Logged out.' });
  });
});
```

What this does:

- Destroys the session (forgets who was logged in)
- Sends success message

Part 9: Protected Routes

Authentication Middleware

```javascript
function requireAuth(req, res, next) {
  if (!req.session.userId) {
    return res.status(401).json({ error: 'Unauthorized' });
  }
  next();
}
```

How middleware works:

- Runs before protected routes
- Checks if user is logged in (has userId in session)
- If not logged in, sends error
- If logged in, calls `next()` to continue to the actual route

Profile Route

```javascript
app.get('/profile', requireAuth, (req, res) => {
  db.get('SELECT id, name, email FROM users WHERE id = ?', [req.session.userId], (err, user) =>
    if (err || !user) return res.status(404).json({ error: 'User not found.' });
    res.json(user);
  });
});
```

What this does:

1. Requires authentication (using middleware)
2. Gets user info from database using session userId
3. Returns user's name and email (but not password!)

Settings Management

```javascript
app.post('/settings', requireAuth, (req, res) => {
  const { storeExcelConversions } = req.body;
  if (typeof storeExcelConversions === 'undefined') {
    return res.status(400).json({ error: 'Missing setting.' });
  }
  db.run(
    `INSERT INTO settings (userId, storeExcelConversions) VALUES (?, ?)
     ON CONFLICT(userId) DO UPDATE SET storeExcelConversions=excluded.storeExcelConversions`,
    [req.session.userId, storeExcelConversions ? 1 : 0],
    function(err) {
      if (err) return res.status(500).json({ error: 'Failed to update settings.' });
      res.json({ success: true, storeExcelConversions: !!storeExcelConversions });
    }
  );
});

app.get('/settings', requireAuth, (req, res) => {
  db.get('SELECT storeExcelConversions FROM settings WHERE userId = ?', [req.session.userId], (
    if (err) return res.status(500).json({ error: 'Failed to fetch settings.' });
    res.json({ storeExcelConversions: !!(row && row.storeExcelConversions) });
  });
});
```

Settings Routes:

- POST: Updates user's settings (creates new record or updates existing)
- GET: Retrieves user's current settings

Part 10: Server Startup and Error Handling

Starting the Server

```javascript
app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

What this does:

- Starts the web server on port 3000

- Prints confirmation message
- Server is now ready to accept requests

Global Error Handler

```javascript
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).send('Server error');
});
```

Error handling:

- Catches any unhandled errors
- Logs error to console
- Sends generic error message to user

404 Handler (Page Not Found)

```javascript
app.use((req, res, next) => {
  res.setHeader('X-Content-Type-Options', 'nosniff');
  res.setHeader('X-Frame-Options', 'DENY');
  res.setHeader('X-XSS-Protection', '1; mode=block');
  res.setHeader('Referrer-Policy', 'no-referrer');
  res.setHeader('Permissions-Policy', 'geolocation=(), microphone=()');
  res.setHeader('Strict-Transport-Security', 'max-age=31536000; includeSubDomains');
  res.setHeader('Content-Type',"text/html");
  res.status(404).send(`
    <!DOCTYPE html>
    <html>
      <head><title>404 Not Found</title>
      <link rel = 'stylesheet' href = '/style.css'>
      </head>
      <body>
        <h1>Sorry, can't find that!</h1>
        <p>The page you requested does not exist.</p>
      </body>
    </html>
  `);
})
```

Security Headers:

- `X-Content-Type-Options: nosniff` = Prevents browser from guessing file types
- `X-Frame-Options: DENY` = Prevents page from being embedded in frames
- `X-XSS-Protection` = Enables browser's XSS protection
- `Referrer-Policy` = Controls what referrer info is sent
- `Permissions-Policy` = Restricts access to browser features
- `Strict-Transport-Security` = Forces HTTPS connections

404 Page:

- Shows custom error page when user visits non-existent URL
- Includes CSS styling
- User-friendly error message

Summary

Your server.js creates a full-featured web application that:

1. **Handles file uploads** - Users can upload Excel files
2. **Converts data** - Excel → JSON, PDF, or SQL
3. **Manages users** - Registration, login, logout
4. **Stores data** - SQLite database for users and settings
5. **Provides security** - Password hashing, sessions, security headers
6. **Handles errors** - Graceful error handling and 404 pages

The application follows good practices like input validation, error handling, and security measures. It's a solid foundation that could be extended with more features!