Case Study 1: Secure File Upload and Billing System

Scenario:
A SaaS company wants to build a **secure file upload and billing system**.

Requirements:

1. Upload user files and store them on the server (use `fs` and streams).
2. Keep track of uploaded files per user (using modules).
3. Generate a **billing summary** for storage usage.
4. Securely hash files using `crypto`.
5. Use **async operations** and **non-blocking I/O**.
6. Serve a simple HTTP API to get the billing info.
7. Demonstrate **event loop behavior**, **custom modules**, and **dynamic imports**.

Project Structure

```
node-case-study/
├─ modules/
│   ├─ user.js          // User and file storage logic
│   ├─ billing.js       // Billing logic
│   ├─ config.js        // Config values
│   └─ index.js         // Re-export modules
├─ uploads/             // Folder to store uploaded files
└─ server.js            // HTTP server
```

Step-by-Step Implementation: Secure File Upload and Billing System

Step 1: Set Up Project Structure

1. Create a main project folder, e.g., `node-case-study`.
2. Inside the project, create a folder named `modules` to store all the modules.

3. Create another folder named `uploads` where user files will be saved.
4. Create a main server file, e.g., `server.js`.
5. Inside `modules/`, create separate files for different functionality:
    - `user.js` → user and file handling logic
    - `billing.js` → billing calculation logic
    - `config.js` → configuration constants
    - `index.js` → re-export all modules for easier imports

Step 2: Create User Module (`user.js`)

Purpose: Keep track of users and their uploaded files.

1. Create a `User` class with the following properties:
    - `name` → the user's name
    - `email` → the user's email
    - `files` → an array to store uploaded files for the user
2. Add a method `uploadFile(fileName, content)` that:
    - Saves the file to the `uploads` folder
    - Generates a **hash** of the file content for security
    - Stores file metadata (name, size, hash) in the `files` array
3. Create a global array `users` to keep track of all user objects.

Concepts Used: Classes, arrays, non-blocking file operations, crypto for hashing, module exports.

Step 3: Create Billing Module (`billing.js`)

Purpose: Calculate storage usage and billing for each user.

1. Create a function `calculateBill(user)` that:
    - Loops through all files uploaded by a user
    - Adds up the total size
    - Multiplies by a price per KB to compute the bill

2. Create a function `billingSummary()` that:
    - Loops through all users
    - Returns a summary for each user including:
        - Name
        - Email
        - Number of files uploaded
        - Total bill amount

Concepts Used: Functions, arrays, loops, module exports.

Step 4: Create Configuration Module (`config.js`)

Purpose: Store constants and configuration values for the application.

1. Add constants such as:
    - `PORT` → which port the HTTP server will listen on
    - `MAX_FILE_SIZE` → maximum allowed file size for uploads
2. Export these constants so they can be used in other modules.

Concepts Used: Constants, module exports, ES modules (or CommonJS).

Step 5: Re-export Modules (`index.js`)

Purpose: Simplify imports in the main server file.

1. Import the `User` class, `users` array, `calculateBill`, `billingSummary`, and configuration constants from their respective files.
2. Export them all from `index.js`.

Concepts Used: Module re-export, organized code structure.

Step 6: Create the HTTP Server (`server.js`)

Purpose: Serve API endpoints and handle user interactions.

1. Import necessary modules from `index.js`.
2. Create a few sample user objects.

3. Call `uploadFile()` for each user to simulate file uploads.
4. Create an HTTP server:
    - If the request URL is `/billing`, return the billing summary as JSON.
    - Otherwise, return a welcome message.
5. Start the server and listen on the configured port.

Concepts Used: HTTP server, routing, JSON response, console logging, asynchronous operations.

Step 7: Demonstrate Event Loop and Non-Blocking I/O

1. File uploads are saved asynchronously, so Node.js continues executing other code without waiting.
2. Event loop handles multiple asynchronous callbacks (file writes, HTTP requests).
3. Use `console.log` to observe the order of execution.

Concepts Used: Event loop, async callbacks, non-blocking I/O.

Step 8: Secure File Handling

1. Generate a hash (SHA256) for every uploaded file to ensure file integrity.
2. Store file metadata along with hash in the user's file list.

Concepts Used: Crypto module, file security, hashing.

Step 9: Billing Logic

1. Use the `billingSummary()` function to calculate bills for all users.
2. Billing is based on **total file size uploaded by each user**.
3. Display summary either in console or via HTTP API.

Concepts Used: Arrays, loops, data aggregation, module usage.

Step 10: Testing and Debugging

1. Run the server using `node server.js`.
2. Open `http://localhost:<PORT>/billing` in a browser to see billing info.

3. Check `uploads/` folder to verify files are stored.
4. Use `console.log()` to debug flow and check uploaded file details.
5. Optional: Use `node --inspect server.js` and Chrome DevTools for step debugging.

Concepts Used: Debugging, profiling, testing async operations.

Step 6: Sample Output

Console Output (uploads & logs):

```
File file1.txt uploaded by Alice
File file2.txt uploaded by Bob
Server running on port 4000
```

HTTP Request: `http://localhost:4000/billing`

JSON Response:

```
[
  {
    "name": "Alice",
    "email": "alice@example.com",
    "totalFiles": 1,
    "bill": 1.2
  },
  {
    "name": "Bob",
    "email": "bob@example.com",
    "totalFiles": 1,
    "bill": 2.0
  }
]
```

Key Concepts Demonstrated

1. **Node.js Architecture:**
    - Async file operations → non-blocking I/O.
    - Event loop handles multiple callbacks.
2. **Modules:**

- CommonJS + ES modules.
- Index file re-exports → clean imports.

3. **Standard Library Usage:**
   - `fs` → write files asynchronously.
   - `http` → create server.
   - `path` → construct file paths.
   - `crypto` → SHA256 hashing.
4. **Debugging & Profiling:**
   - Use `console.log` for basic logs.
   - `node --inspect server.js` → step debugging in Chrome DevTools.
5. **Scenario-Based Learning:**
   - Real-world SaaS system: file upload + billing + API.