Working with Embedded Documents and Arrays in MongoDB

In the final part of last week's (Week 4) lab, you were asked to insert the following student data into the **Students** collection within the **University** database to carry out further experiments.

UoB	**Name**	**Age**	**Course**	**House No**	**Street**	**Town**	**Hobbies**
123456	Alice Johnson	22	Computer Science	45	High Street	Birmingham	Traveling
876543	Bob Smith	20	Mathematics	12B	Baker Street	London	Playing Chess, Painting
234567	Clara Lee	23	Physics	78	King Street	Manchester	Photography, Hiking
345678	David Brown	27	Engineering	14A	Queen Road	Leeds	Football, Gaming
456789	Emma Wilson	25	Business Management	34	Church Lane	Liverpool	Singing, Traveling
567890	Frank Adams	31	Economics	56	Bridge Street	Glasgow	Cooking, Reading
678901	Grace Thomas	19	Biology	22	Elm Road	Oxford	Swimming, Painting, Reading
789012	Henry Walker	32	Chemistry	9	Maple Avenue	Cambridge	Cycling, Writing
890123	Ivy Carter	27	Law	50	Park Lane	Leeds	Volunteering, Gardening
901234	Jack White	25	History	31	Oak Street	Manchester	Running, Cooking

There are multiple ways to insert data into a MongoDB collection. Using the **Mongo shell**, you can either:

- Insert each row as a single document using the `insertOne(document)` command, or
- Insert multiple documents at once using `insertMany([d1, d2, …, dn])`.

Alternatively, using **MongoDB Compass**, you can directly import datasets in **CSV** or **JSON** format.

However, note that the **Hobbies** field in our dataset is a *multi-valued attribute* (i.e., it contains comma-separated values). When importing from a CSV file, this field will be stored as a single string rather than as an array of individual values, which is not suitable for MongoDB's document structure.

To address this, it is preferable to **convert the dataset into the appropriate JSON format** before importing it using Compass.

A correctly formatted JSON version of the dataset, named **`students.json`**, has been provided under Week 4's materials.

- **Task 1:** Download the **`students.json`** file and open it in a text editor to examine how the dataset is structured and formatted.
- **Task 2:** Reflect on the following questions:
    1. How can a CSV file be transformed into this JSON structure?
    2. Is there a way to automate this transformation process?

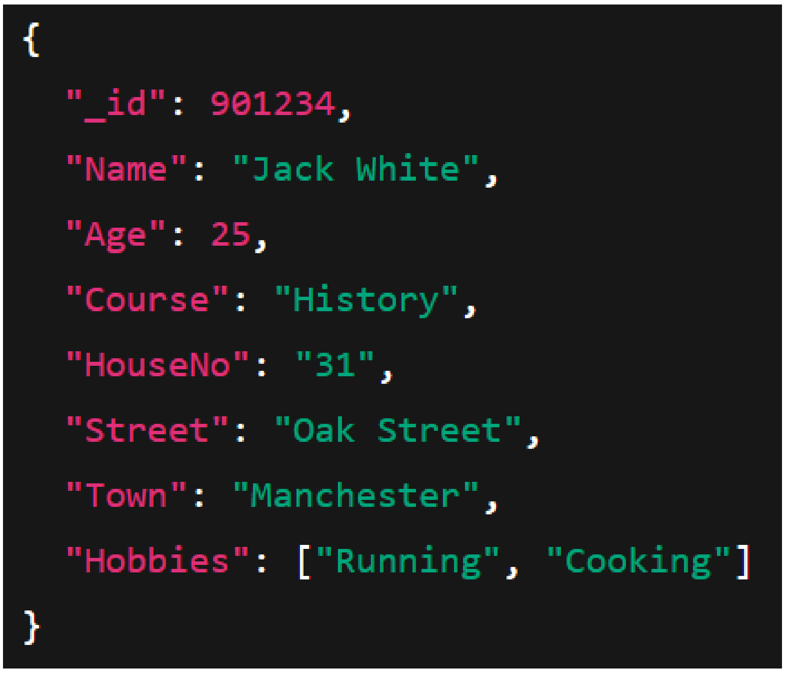
In this lab, first you will explore one possible solution using a short **Python script** to automate this conversion and gain a better understanding of how structured data (CSV) can be translated into semi-structured formats (JSON) suitable for MongoDB.

The provided Python script (as seen below), **`CSV_to_JSON_flat.py`**, reads the CSV file **`students.csv`** (available in Canvas) and generates a JSON file named **`students.json`**, containing a flat, non-embedded representation of the data.

Carefully review the script to understand how each CSV column is mapped to a corresponding field in the JSON document.

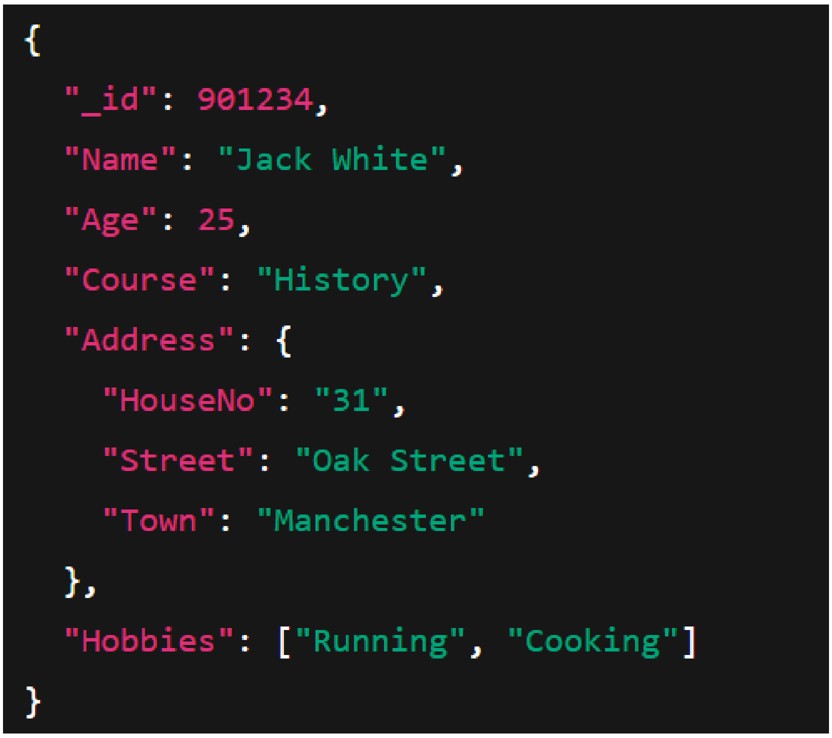
When you open the generated JSON file, you will notice that each record is stored as a flat document (as seen in the example below), where all fields (such as HouseNo, Street, and Town) appear at the same hierarchical level. However, in real-world applications, related information is often grouped together for clarity and efficiency.

```python
import csv
import json

# Input and output file names
csv_file = "students.csv"
json_file = "students.json"

# List to hold all records
data = []

# Open and read the CSV file
with open(csv_file, encoding='utf-8') as f:
    reader = csv.reader(f)
    headers = next(reader)  # Skip header row if present

    for row in reader:
        # Each row: UoB, Name, Age, Course, House No, Street, Town, Hobbies
        student = {
            "_id": int(row[0]),  # Use student ID as _id
            "Name": row[1],
            "Age": int(row[2]),
            "Course": row[3],
            "HouseNo": row[4],
            "Street": row[5],
            "Town": row[6],
            "Hobbies": [h.strip() for h in row[7].split(",")]  # Convert to list
        }
        data.append(student)

# Write list of dictionaries to JSON file
with open(json_file, "w", encoding='utf-8') as f:
    json.dump(data, f, indent=2)

print(f"Successfully converted {csv_file} to {json_file}")
```



```json
{
  "_id": 901234,
  "Name": "Jack White",
  "Age": 25,
  "Course": "History",
  "HouseNo": "31",
  "Street": "Oak Street",
  "Town": "Manchester",
  "Hobbies": ["Running", "Cooking"]
}
```

In this exercise, your goal is to transform the flat structure into a **hierarchical (embedded) format**, where address-related fields are grouped into a single embedded sub-document called **`address`**, as illustrated in the example below.



```
{
  "_id": 901234,
  "Name": "Jack White",
  "Age": 25,
  "Course": "History",
  "Address": {
    "HouseNo": "31",
    "Street": "Oak Street",
    "Town": "Manchester"
  },
  "Hobbies": ["Running", "Cooking"]
}
```

Tasks

1. **Modify the Python Script**

   Adapt the provided **`CSV_to_JSON_flat.py`** script so that it reads the CSV file and generates a new JSON file containing embedded documents (as shown above).

2. **Import the JSON File**

   Import the generated JSON file into a MongoDB collection named **`newStudents`** within the **`University`** database.

3. **Query Performance Without Indexes**

   Use the **`explain()`** method (see the lecture note for this) to examine how MongoDB executes the following queries **before** creating any indexes:

   - Find all students older than 21.
   - Retrieve the first 10 students over the age of 25.
   - Find all students enrolled in the *Computer Science* course who are older than 22.

   Observe and record key execution statistics such as:

  - `nReturned`
  - `totalDocsExamined`
  - `executionTimeMillis`

4. **Create Indexes**

   Create indexes to improve query efficiency:

   - A single-field index on **`Age`**: **`db.newStudents.createIndex({ Age: 1 })`**
   - A compound index on **`Course`** and **`Age`**:

     ```
     db.newStudents.createIndex({ Course: 1, Age: 1 })
     ```

5. **Query Performance With Indexes**

   Re-run the same queries using **`explain()`** and compare the results to those obtained before indexing.

   Pay particular attention to how indexing affects:

   - The number of documents examined
   - Query execution time

6. **Analysis**

   Briefly discuss how indexing influences query performance in MongoDB. Which queries benefit most, and why?

Embedding Documents in MongoDB as Arrays of Documents

In the previous exercises you learned how to embed a simple sub-document (e.g. `address`) and how to store arrays of strings (e.g. `hobbies`). In this part, you will take the next step and store **arrays of documents** inside a student document — i.e. each student will have a courses field where each element is a small document with `courseName`, `instructor`, and `credits`.

Before you begin — Inspect the CSV

Download **`students_courses.csv`** (available on Canvas) and open it in a text editor or spreadsheet program. Carefully inspect the rows and consider the following:

1. **How many unique students are there?**
2. **Is there redundant data? Why does it appear?**

   You should notice that student fields (name, address, program, hobbies, etc.) repeat across multiple rows when a student is enrolled in multiple courses. This redundancy is typical of a *relational/row-wise* export where each student–course pair is one row. In document databases we often remove this redundancy by embedding the repeated (course) data as an array of documents inside the single student document.

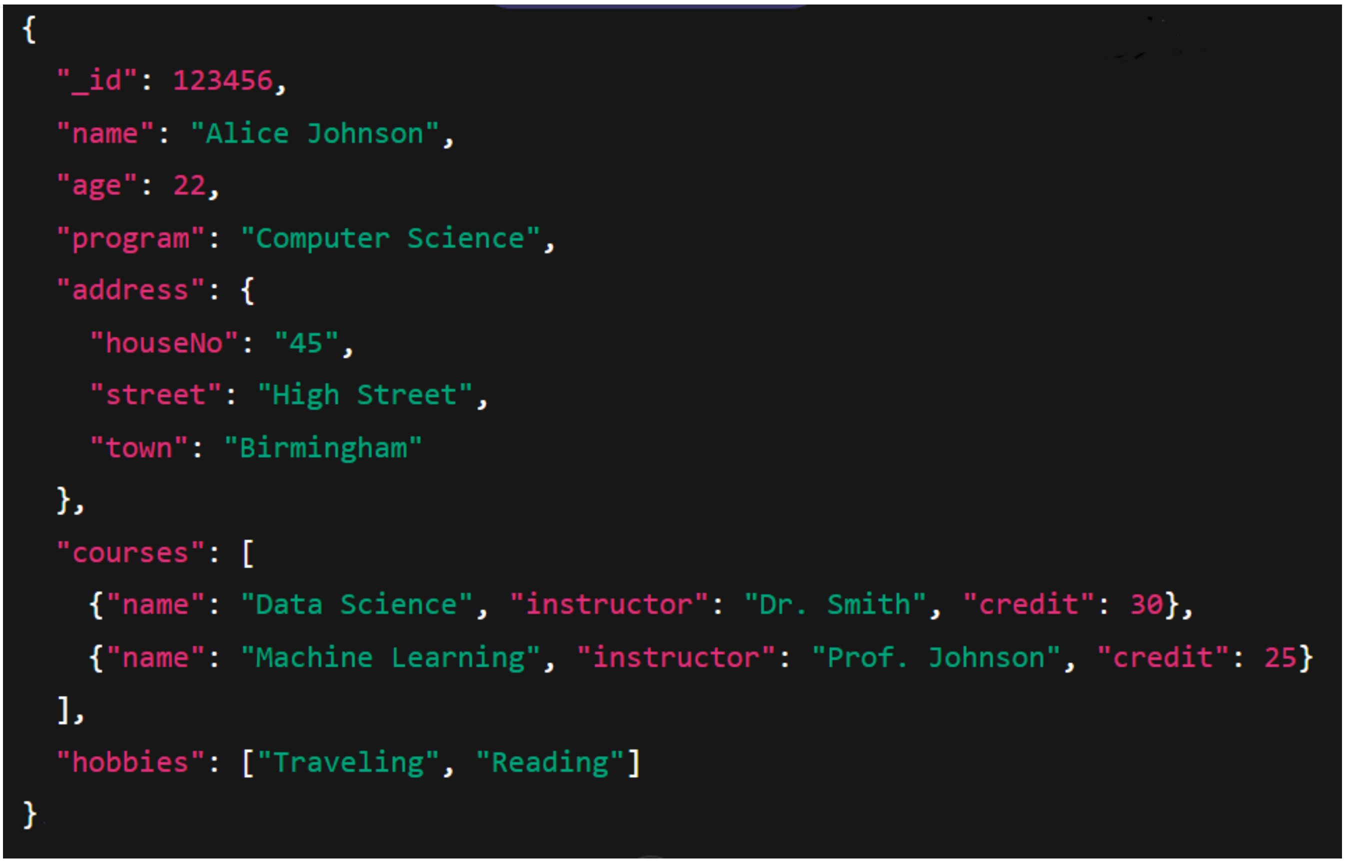
Write down your answers to (1) and (2) — these observations motivate the transformation in this lab.

Why embed courses as an array of documents?

- **Semantics:** A student *has many* courses; courses are tightly related to student records (one-to-few or one-to-many where number of courses per student is moderate).
- **Performance:** Reads that require student + their courses are fast because all data is in one document.
- **Simplicity:** Easier to express queries like “students taking *Data Analytics*” without joining collections.

Trade-off: if course details must be updated centrally across many students (and the number of students per course is huge), a referenced model might be preferable.

After transformation each student document should look like this:



```
{
  "_id": 123456,
  "name": "Alice Johnson",
  "age": 22,
  "program": "Computer Science",
  "address": {
    "houseNo": "45",
    "street": "High Street",
    "town": "Birmingham"
  },
  "courses": [
    {"name": "Data Science", "instructor": "Dr. Smith", "credit": 30},
    {"name": "Machine Learning", "instructor": "Prof. Johnson", "credit": 25}
  ],
  "hobbies": ["Traveling", "Reading"]
}
```

Tasks

1. Create the collection

Create (or switch to) the `University` database and create a **`Students_Courses`** collection.

2. Convert & Insert the data

Transform the `students_courses.csv` (one row per student–course pair) into JSON where each student appears once and courses is an array of documents. Insert these documents into the `Students_Courses` collection.

3. Perform queries (practice querying arrays of documents

a. Find all students enrolled in a course named **"Data Analytics"**

b. Find all students who have **"Dr. Adams"** as instructor for any course

c. Find all students who are enrolled in a course with **credits < 25**

4. Modify embedded data

Change every course record where the instructor is "**Dr. Smith**" to "**Prof. Johnson**" across all students. Use `updateMany()` with `arrayFilters` to target matching array elements.

Optional Extension Task

Write a **Python script** that:

- Reads a CSV file with students and courses (one row per student–course pair).
- Converts it into JSON format with:
  - Embedded address sub-document
  - Array of course documents
  - Array of hobbies
- Saves it as a MongoDB importable JSON file.