### ****Detailed Report on CocktailDB****

#### ****Project Overview****

The aim of this project was to fetch cocktail data from an online API, clean it up, and store it in a database. The goal was to ensure that the data is structured properly, making it easy to query and use later. The key steps included:

1. **Fetching data from the API.**
2. **Cleaning and organizing the data into a useful format.**
3. **Handling missing or inconsistent data.**
4. **Saving the cleaned data into a SQLite database** for future use.

#### ****1. Data Ingestion: Fetching Cocktail Data from the API****

**Why This Approach?**  
The data was pulled from the public API provided by [TheCocktailDB](https://www.thecocktaildb.com/api/json/v1/1/search.php?). By querying the API for cocktails starting with every letter of the alphabet (from A to Z), we ensured that we captured all available cocktails. For example, we queried f=a for drinks starting with 'A', f=b for drinks starting with 'B', and so on.

**Why Use This Loop?**

* **Efficiency**: The API only returns a limited number of results at a time, so looping through all letters ensures we don’t miss any cocktails.
* **Scalability**: The number of drinks available per letter might vary, so this loop ensures comprehensive coverage without overloading the system.

**Why It Works Well**:

* **Keeps things modular**: The code is simple and easy to test by querying one letter at a time.
* **Quick and efficient**: We fetch all data without retrieving unnecessary information.

#### ****2. Data Transformation: Cleaning and Normalizing the Data****

**Why We Do This**  
The raw data fetched from the API was messy and not structured well. We cleaned up the data and organized it into two tables:

* **Drinks Table**: Contains general info about each drink (e.g., name, category, instructions).
* **Ingredients Table**: Contains ingredients and their measurements for each drink.

**Normalization**

* By placing the ingredients in a separate table, we avoid repeating ingredient data each time a drink appears. This prevents redundancy.

**Why Normalize?**

* **No Redundancy**: Without normalization, ingredient data would be repeated for every cocktail that shares the same ingredient (e.g., “lime” or “vodka”). By normalizing, we store ingredient info only once.
* **Easier Queries**: If we want to find all drinks containing a specific ingredient (e.g., “lime”), querying the Ingredients table is faster and more efficient than searching through all drinks.

#### ****3. Handling Edge Cases****

##### **Edge Case 1 - Mismatched Ingredients and Measures**

**Why It's Important**  
Not all cocktail data is perfect. Some drinks might be missing ingredients, or the ingredients and their measurements might not align properly. Our job is to ensure that we don't store incomplete or inconsistent data in the database.

**Edge Case 1:**

* **If Ingredients Don’t Match Measures**: If there are more ingredients than measures (or vice versa), we skip that cocktail entirely. We want to ensure that the data we store in the database is valid and complete.

**Why This Makes Sense**:

* **Keeps Data Clean**: This ensures that only complete and valid drink data is saved into the database.
* **Makes It More Robust**: If the API returns messy data (e.g., mismatched ingredients or missing measurements), this approach ensures the script won't break or insert invalid data into the database.

##### **Edge Case 2 - Conversion to mL for Measurements**

**Why It's Important**  
Some ingredients in cocktails are listed with measurements that are not directly numerical or are non-standard (e.g., "frozen," "dash," "garnish"). These cannot be converted to milliliters (mL) and could cause errors in the data processing.

**Edge Case 2:**

* **Non-Numeric and Irregular Measures**: Measurements like "frozen," "dash," or "garnish" don't have a numeric value and cannot be converted directly to milliliters. To handle this:
  + The script checks if the measurement contains such values.
  + If found, these values are returned as they are, and no conversion is performed.
  + This ensures that the non-numeric values are preserved, and no error is raised during the conversion process.

**Why This Makes Sense**:

* **Prevents Errors**: This ensures that the script doesn’t crash or produce incorrect conversions when it encounters non-numeric or non-standard measures.
* **Keeps Data Integrity**: By handling these edge cases gracefully, the data remains accurate, and the process becomes more reliable.

#### ****4. Storing the Data in SQLite****

**Why SQLite?**

* **Lightweight**: SQLite does not require a complex server setup and is suitable for small projects.
* **Portable**: The database is stored in a single file, making it easy to share and move around.
* **Simple**: SQLite is ideal for small to medium-sized projects where structured data storage is needed without a heavy database system.

We created two tables in SQLite:

* **Drinks Table**: Stores basic drink information such as the drink's name, instructions, category, and more.
* **Ingredients Table**: Stores ingredient names and measurements for each drink.

**Why Use SQLite?**

* **No Overhead**: It’s easy to set up and does not require a server.
* **Portable**: The database is stored as a single file, which makes it easy to share or back up.
* **Perfect for this project**: SQLite is suitable for this project, where simple data storage is required.

**Data Integrity**:  
Once the data was cleaned, it was committed to the SQLite database using if\_exists='replace'. This replaced any existing data in the database, ensuring that we always work with fresh data. If needed, we could change this to if\_exists='append' to add new data without overwriting the existing records.

### ****Conclusion****

The code was designed with the following goals:

* **Simple**: It fetches data from the API and stores it in a clean, usable format.
* **Efficient**: The data is normalized to reduce redundancy and simplify queries.
* **Robust**: The code handles edge cases such as mismatched ingredients, non-numeric measurements, and irregular values without breaking the process.
* **Portable**: SQLite makes the data easy to store and share in a single file.