ATLAN ASSIGNMENT

TASK - 1:

1. One of the clients wanted to search for slangs (in local language) for an answer to a text question on the basis of cities (which was the answer to a different MCQ question)

APPROACHES:

The problem statement describes a scenario where clients want to search for slang in the local language based on cities, which are answers to different multiple-choice questions (MCQs). Here are several potential ways to approach and solve this problem:

1. Slang Mapping in the Data Store:

- Solution:
  - Maintain a mapping table in the data store that associates each city with its corresponding slang in the local language.
- Pros:
  - Simple and direct mapping.
  - Allows for efficient and fast retrieval during searches.
- Cons:
  - Requires manual entry and maintenance of the slang-city mapping.

2. External API Integration:

- Solution:
  - Integrate with external APIs or databases that provide slang information for cities.
- Pros:
  - Leverages existing resources.
  - Reduces the need for manual maintenance.
- Cons:
  - Dependency on external services.
  - May have associated costs.

SYSTEM DESIGN:

DATA BASE LOOKUP:

- Data Model:
  - Maintain a table in the data store with columns for city, local language slang, and any other relevant metadata.

- Data Maintenance:
  - Regularly update the mapping table with new slang and cities.
- Search Mechanism:
  - Perform a simple lookup in the mapping table based on the selected city in a response.

2. EXTERNAL API INTEGRATION:

- API Integration:
  - Integrate with external APIs or databases that provide slang information for cities.
- Caching:
  - Implement caching mechanisms to reduce the number of API calls and improve performance.
- Fallback Mechanism:
  - Include a fallback mechanism in case the external API is unavailable.

CHOOSEN APPROACH:

External Api Integration

CODEBASE:

Clearly specified in the readme file

END OF TASK - 1

TASK – 2:

2. A market research agency wanted to validate responses coming in against a set of business rules (eg. monthly savings cannot be more than monthly income) and send the response back to the data collector to fix it when the rules generate a flag

Middleware Validation Function:

- The middleware function validateData checks if the savings are more than income and if the mobile number is a valid 10-digit number.
- Route for Insertion:
    - The validateNew route utilizes the validateData middleware to ensure data integrity before inserting a new client's details into the database.
- Route to Validate All Records:
    - The validateAll route checks for all records where savings are greater than income and returns either a message indicating all records are valid or a list of invalid records.

Pro/Con Analysis:

- Pros:
    - Ensures data integrity during insertion.
    - Provides a route to validate all records for potential missed validations.
    - Clear separation of concerns with the middleware function.
- Cons:
    - Limited to basic validation rules; more complex rules might require additional middleware.
    - Potential performance impact with large datasets in the validateAll route.
    - This approach strikes a balance between simplicity and effectiveness, ensuring that individual insertions are validated and providing a mechanism to catch and address any missed validations. The middleware approach is modular, making it easy to extend and maintain

SYSTEM DESIGN:

- **Client-Server Architecture:**
    - The system follows a client-server architecture where clients (e.g., web browsers) make requests to the server, and the server processes those requests.
    - Web Server (Node.js + Express):
    - Node.js with Express is used as the web server to handle HTTP requests and responses.
- **Database (PostgreSQL):**
    - PostgreSQL is chosen as the relational database to store client income data.
- **Middleware:**
    - Middleware functions, such as validateData, are incorporated into the Express application to validate data before insertion.
- **Routes:**
    - Express routes (/validateNew, /validateAll) define the API endpoints for inserting new client data and validating all records.
    - Client (Web Browser or AJAX):
    - Clients interact with the system through a web browser or AJAX requests.

Sequence of Operations:

- Insertion with Validation:
    - Client sends a POST request to /validateNew with client details.
    - Middleware (validateData) checks data integrity.
    - If valid, the server inserts data into the PostgreSQL database.
    - The server responds with the newly inserted client data.
    - Validation of All Records:
    - Client sends a GET request to /validateAll.
    - The server queries the PostgreSQL database for records where savings are greater than income.
    - If all records are valid, the server responds with a message.
    - If there are invalid records, the server responds with the list of invalid records.

END OF TASK - 2

TASK 3:

3. A very common need for organizations is wanting all their data onto Google Sheets, wherein they could connect their CRM, and also generate graphs and charts offered by Sheets out of the box. In such cases, each response to the form becomes a row in the sheet, and questions in the form become columns.

APPROACHES:

1. Google Sheets API Integration:

Overview:

- How it works: Utilize the Google Sheets API to programmatically interact with Google Sheets.
- Implementation Steps:
- Authenticate with Google Sheets API using OAuth.
- Use the API to create a new sheet or update an existing sheet.
- Map form responses to rows in the sheet.

Pros:

- Direct integration with Google Sheets
- Fine-grained control over the data mapping and updates.

Cons:

- Requires programming knowledge for API integration.
- Maintenance and potential changes in the Google Sheets API.

2. Third-Party Integration Services:

Overview:

- How it works:
  - Use third-party integration platforms like Zapier, Integromat, or Tray.io to connect the form with Google Sheets.
- Implementation Steps:
  - Create a workflow on the integration platform.
  - Set up triggers to detect new form responses.
  - Define actions to update or append data to Google Sheets.

Pros:

- No coding skills required.
  Pre-built connectors for many popular apps, including Google Sheets.

Cons:

- Limited customization compared to direct API integration.
- May have usage limitations in free plans.

COOSEN APPROACH:

- **fetchRecordsFromDatabase Function:**
  - Establishes a connection to the SQLite database (students2.db).
  - Performs a SELECT query to retrieve all records from the "students" table.
  - If an error occurs during the query, it logs the error, invokes the callback with the error, and closes the database connection.
  - If the query is successful, it invokes the callback with no errors and the fetched rows, then closes the database connection.

- **/getCSV Route:**
  - Set up to handle a GET request to the "/getCSV" endpoint.
  - Calls the fetchRecordsFromDatabase function using myDatabaseRef.
  - If there's an error during the database fetch, it responds with a JSON object containing the error details and a custom error message.
  - If records are successfully fetched, it calls myConvertJSONToCSV with the fetched rows and the response object (res).

- **/getCSV Route (GET Method):**
  - A GET request to http://localhost:3000/getCSV triggers the route handler.
  - The route handler fetches records from the SQLite database using fetchRecordsFromDatabase.
  - If successful, it converts the fetched records to CSV format and triggers a download.

WHY DID I CHOOSE THIS APPROACH:

- Error Handling:
  - the code includes error handling for both the database query and the subsequent CSV conversion. This ensures that potential issues are caught and appropriately handled.
- Asynchronous Design:
  - The use of asynchronous functions (db.all) allows for non-blocking operations, ensuring that the application remains responsive even during potentially time-consuming database queries.
- Reusability:
  - The fetchRecordsFromDatabase function can be reused in other parts of your application or routes, promoting code reusability.

- Clean Route Handling:
  - The route handler focuses on handling HTTP requests and delegating database interactions to the fetchRecordsFromDatabase function. This promotes clean and readable route handling logic

This approach adheres to best practices in terms of modularity, error handling, and asynchronous design, making it a reasonable choice for fetching records from an SQLite database and exporting them in CSV format.

SYSTEM DESIGN:

- User makes a GET request to /getCSV.
- /getCSV route calls fetchRecordsFromDatabase.
- fetchRecordsFromDatabase fetches records from the SQLite database.
- If successful, it calls myConvertJSONToCSV with the fetched rows and response object.
- myConvertJSONToCSV converts JSON to CSV and triggers a download.

END OF TASK - 3

TASK 4:

4. A recent client partner wanted us to send an SMS to the customer whose details are collected in the response as soon as the ingestion was complete reliably. The content of the SMS consists of details of the customer, which were a part of the answers in the response. This customer was supposed to use this as a “receipt” for them having participated in the exercise.

APPROACHES:

1. SMS API Integration:

- Approach:
- Integrate with a third-party SMS service provider that offers an API for sending SMS messages.
- When data ingestion is complete, trigger a request to the SMS API with the relevant customer details.
- The SMS provider will handle the actual sending of the SMS.

2. USING NODEMAILER LIBRARY: (Middleware for Sending Email)

mySendEmail Function:

- Functionality:
  - This function uses the nodemailer library to send an email.
  - It configures the email transport options, such as the SMTP server details and authentication credentials.
  - The email content is specified using parameters (emailSubject and emailMessage).
  - The transporter.sendMail method sends the email with the provided options.

Route for Sending Email:

- Middleware Function (SMS):
  - This middleware function is intended for sending an email.
  - It extracts the email subject and message from the request body.
  - Calls the mySendEmail function to send the email.
  - Responds with a success message.

Route Configuration:

- Route (POST Method):
  - The route is defined for the endpoint /sendmessage using the POST method.
  - Upon receiving a POST request to this endpoint, it triggers the SMS middleware to send an email.

CHOOSEN APPROACH:

Nodemailer Middleware Approach.

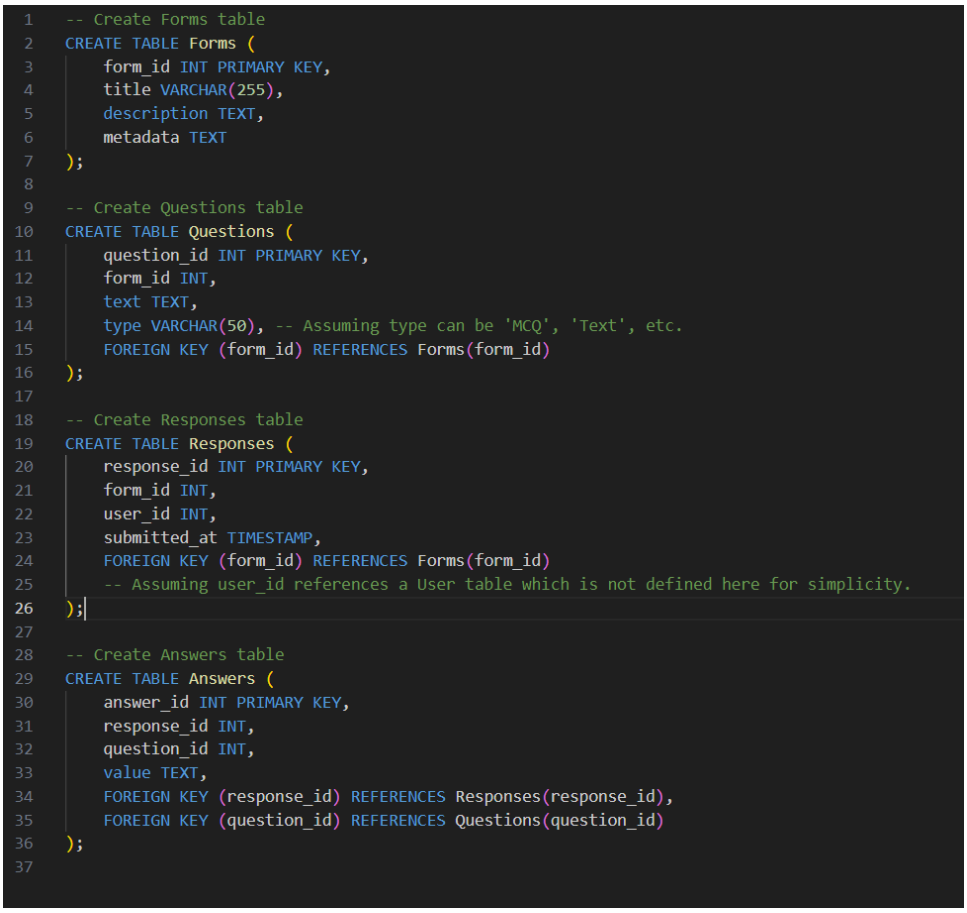
WHY DID I CHOOSE NODE MAILER APPROACH:

- Modularity:
  - The use of middleware separates email-sending logic, improving code organization.
- Integration:
  - Easily integrates with external email services using the nodemailer library.
- Error Handling:
  - Basic error handling is included for identifying and resolving issues.

SYSTEM DESIGN:

END OF TASK - 4

SAMPLE SCHEMATIC OF HOW THE DATABASE STORES THE FORM'S DATA:



```sql
-- Create Forms table
CREATE TABLE Forms (
    form_id INT PRIMARY KEY,
    title VARCHAR(255),
    description TEXT,
    metadata TEXT
);

-- Create Questions table
CREATE TABLE Questions (
    question_id INT PRIMARY KEY,
    form_id INT,
    text TEXT,
    type VARCHAR(50), -- Assuming type can be 'MCQ', 'Text', etc.
    FOREIGN KEY (form_id) REFERENCES Forms(form_id)
);

-- Create Responses table
CREATE TABLE Responses (
    response_id INT PRIMARY KEY,
    form_id INT,
    user_id INT,
    submitted_at TIMESTAMP,
    FOREIGN KEY (form_id) REFERENCES Forms(form_id)
    -- Assuming user_id references a User table which is not defined here for simplicity.
);

-- Create Answers table
CREATE TABLE Answers (
    answer_id INT PRIMARY KEY,
    response_id INT,
    question_id INT,
    value TEXT,
    FOREIGN KEY (response_id) REFERENCES Responses(response_id),
    FOREIGN KEY (question_id) REFERENCES Questions(question_id)
);
```