Atlan Backend Challenge

Report

Abstract

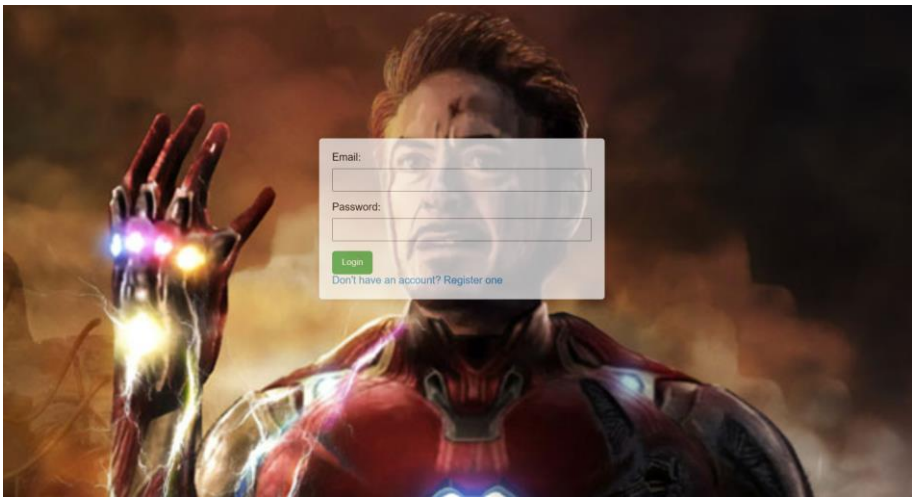
This report explores user authentication, data validation, and middleware solutions within a web application. It emphasizes the importance of user-friendly interfaces and secure data handling. The assessment covers approaches like database-centric SQL, third-party API integration, and middleware for data validation, export to CSV, and SMS integration. The findings offer insights into scalability, security, and performance considerations, guiding improvements for a more robust web application framework.

Pratyush Khare

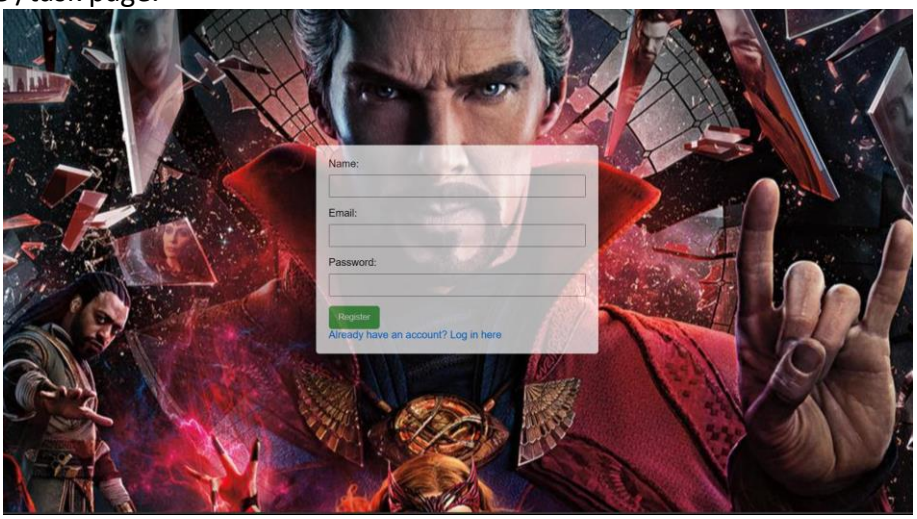
pratyushk824.786@gmail.com

User Authentication (Login and Registration)

- Login: The / route serves the login page (login.html). The /login-user endpoint handles login requests, querying the database for user credentials. If the login is successful, it redirects to the /task page.



- Registration: The /register route serves the registration page (register.html). The /register-user endpoint handles user registration, inserting user data into the database. Successful registration redirects to the /task page.



- /task page:

Pros:

- User-Friendly Responses: The server provides clear responses for successful and unsuccessful login and registration attempts, enhancing the user experience.
- Data Validation: The registration endpoint includes basic data validation, checking if all required fields are filled.

Cons:

- Limited Password Security: The implementation lacks advanced password security features such as hashing and salting, making it susceptible to security vulnerabilities.

Problem Statement:

The lifecycle of data collection via this platform does not end with the submission of a response. There is usually some post-submission business logic that the platform needs to support over time. Some real-life examples -

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

   After analyzing the statement, I am assuming that I have to build a middleware or controller to take a word and convert it into a local slang (local language)

   ## Approach 1:

   ## Database SQL approach:

   We create a database to map the words to slang in other languages
   Table: wordslang

	lang_id [PK] character varying (2)	word [PK] character varying (255)	slang character varying (255)
1	HI	Hello	Namaste
2	hi	hello	namaste

   ## Results:

   ## PROS

- Data Persistence:
  Database storage provides persistent storage for slang mappings. Data remains available even if the server is restarted.
- Scalability:
  Can handle a large amount of data and can scale horizontally by adding more database servers.
- Structured Querying:
  SQL queries provide a structured way to retrieve specific data from the database.
- Consistency:
  Ensures data consistency through the use of transactions. Atomicity, Consistency, Isolation, and Durability (ACID) properties are guaranteed.
- Security:
  Can implement access controls and permissions at the database level to control who can access or modify the data.

   ## CONS

- Database Overhead:
  Introduces additional overhead due to database connections, query execution, and data retrieval. This can impact response time.
- Complexity:
  Adds complexity to the overall system architecture. Requires database setup, maintenance, and potential schema modifications.

- Latency:
  Database queries can introduce latency, especially when dealing with large datasets or complex queries.
- Scalability Challenges:
  While databases are scalable, scaling can become complex and expensive as the amount of data and traffic increases.
- Potential for Data Redundancy:
  Data might need to be duplicated in multiple places, leading to potential redundancy. Changes in slang mappings may require updates in multiple records.
- Development Time:
  Setting up and maintaining a database may require additional development time and effort compared to other simpler storage solutions.

Approach 2:

3rd party API approach:

I have used Google translate api in this approach.
I have designed the route in a way that the user will have to provide the word (The word to be converted) and the language of the slang in a query string.

```
// Middleware to find slang in local language using API - Task 1
async function findSlangAPI(req, res) {
    try {
        var word = req.query.word;
        var language = req.query.lang;
        const response = await translate(word, { to: language });
        console.log(response.text);
        var data = {
            status: "Success",
            GivenWord: word,
            SlangLanguage: language,
            ConvertedSlang: response.text,
        };
        // send the response
        res.status(200).json(data);
    } catch (error) {
        res.status(500).json({
            status: "Failed",
            Msg: "Something Went Wrong!",
            Error: error,
        });
    }
}
```

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 Approach:
   Create a sample Postgres based Relation database

```
CREATE TABLE client_income_data (
  client_id SERIAL PRIMARY KEY,
  client_email VARCHAR(255),
  client_name VARCHAR(255),
  income_per_annum INT,
  savings_per_annum INT,
  mobile_number VARCHAR(10)
);
```

Validation in /validateNew:

- Request Handling:
  Handles HTTP POST requests to the /validateNew endpoint.
- Data Validation Middleware (validateData):
  Validates the request body for the following fields: client_email, income_per_annum, savings_per_annum, and mobile_number.
  Uses regular expressions for email and mobile number validation.
  Check if income_per_annum is greater than or equal to savings_per_annum.
- Database Interaction:
  Inserts data into the client_income_data table in the PostgreSQL database using the pool.query method.
  If the validation passes, returns the inserted client data as JSON.
- Error Handling:
  If any validation fails or there is an error during data insertion, it sends a 400 Bad Request or 500 Internal Server Error response.

Validation in /validateAll:

- Request Handling:
  Handles HTTP GET requests to the /validateAll endpoint.
- Database Query:
  Queries the PostgreSQL database to retrieve records from the client_income_data table where savings_per_annum is greater than income_per_annum.
- Response Handling:
  If there are no invalid records, it sends a response indicating that all records are valid.
  If there are invalid records, it sends the details of those records as a response.
- Error Handling:
  If there is an error during the database query, it sends a 500 Internal Server Error response.

Pros:

/validateNew:

- Data Integrity:
  Ensures data integrity by validating individual records before insertion, preventing incorrect or incomplete data from entering the database.
- Security:
  Guards against common data input errors or malicious attempts by validating email and phone number formats.
- Business Logic:
  Implements business logic by ensuring that savings are not greater than income, preventing illogical data.
- Structured Responses:
  Provides structured JSON responses, making it easier for clients to understand the status of their requests.

/validateAll:

- Data Integrity Across the Dataset:
  Validates the entire dataset to ensure that no records violate the condition savings_per_annum > income_per_annum, maintaining overall data integrity.
- Efficiency:
  Efficiently identifies and communicates the presence of invalid records without the need for manual inspection.

Cons:

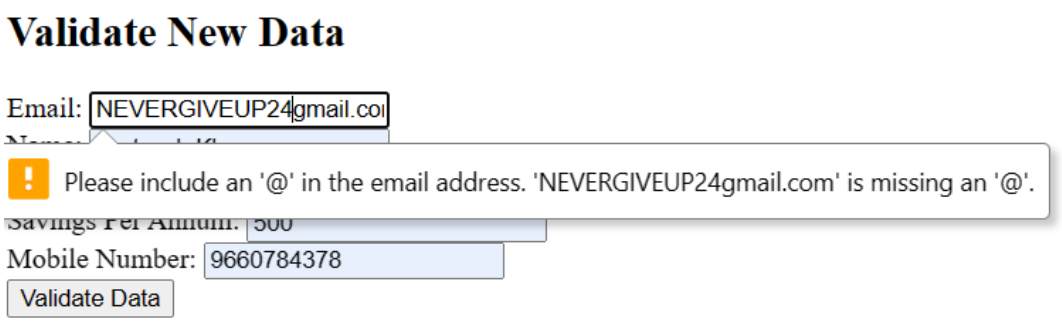
/validateNew:

- Limited Validation:
  The validation focuses on a few specific aspects (email, phone number, and a simple income vs. savings check). Depending on the application requirements, additional validation may be needed.

/validateAll:

- Performance Impact:
  Querying the entire dataset for validation might have a performance impact as the dataset grows. Considerations for optimization may be required.
- Limited Details on Invalid Records:
  The response for invalid records does not provide detailed information about which specific records are invalid. This could make debugging and correction more challenging.
- No Historical Validation:
  The validation is performed at the time of the request and does not maintain historical validation information. This might be a consideration if historical validation data is needed.

Result for invalid Email ID



If every input data is valid then

```
{
   "client_id": 9,
   "client_email": "NEVERGIVEUP24@gmail.com",
   "client_name": "Pratyush Khare",
   "income_per_annum": 600,
   "savings_per_annum": 500,
   "mobile_number": "9660784378"
}
```

Result in the table:

client_id [PK] integer	client_email character varying (255)	client_name character varying (255)	income_per_annum integer	savings_per_annum integer	mobile_number character varying (10)
10	NEVERGIVEUP24@gmail.com	Pratyush Khare	600	500	9660784378
11	pratyushk824.789@gmail.com	Ayush	600	400	9874561230

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.

Middleware Approach:

Create a sample Postgres based Relation database

```
CREATE TABLE client_income_data (
  client_id SERIAL PRIMARY KEY,
  client_email VARCHAR(255),
  client_name VARCHAR(255),
  income_per_annum INT,
  savings_per_annum INT,
  mobile_number VARCHAR(10)
);
```

Route to export and download the CSV file

```
// Middleware to export data in CSV - Task 3
async function exportCSV(req, res) {
  try {
    let data = await pool.query("SELECT * FROM client_income_data");
    data = data.rows;

    var file = fs.createWriteStream("public/data.csv");
    fastcsv
      .write(data, { headers: true })
      .on("finish", function () {
        res.send("<a href='/public/data.csv' download='data.csv' id='download-
link'></a><script>document.getElementById('download-link').click();</script>");
      })
      .pipe(file);

  } catch (err) {
    console.error('Error during CSV export:', err.message);
    res.status(500).send('Internal Server Error');
  }
}
```

Pros:

- Data Export to CSV
  The middleware effectively exports data from the client_income_data table to a CSV file.
- Headers in CSV
  The exported CSV file includes headers, making it more human-readable and providing context to the data.
- Asynchronous Operation
  The data export operation is asynchronous, ensuring that the server remains responsive while the export is in progress.
- Error Handling
  The code includes error handling to manage potential issues during the export process, logging errors, and providing an appropriate response to the client.
- Dynamic File Name
  The file name is dynamic and includes a timestamp, which can be helpful to avoid overwriting files and maintain a record of when the export occurred.

Cons:

- No Check for Empty Data
  The code doesn't check if the result of the database query (data) is empty. If there is no data in the client_income_data table, the exported CSV will still be created, but it will be empty.
- File Overwriting
  If this middleware is triggered multiple times, it will overwrite the existing CSV file without any confirmation or versioning. This may lead to unintentional data loss.
- Inline Script for File Download
  The approach of triggering a file download using an inline script in the response (<script>document.getElementById('download-link').click();</script>) might not work as expected in all scenarios, and it's generally better to rely on standard HTTP headers for file download.
- Limited Configuration

The code does not provide much configuration for the CSV export (e.g., allowing the client to specify columns, filters, etc.).

- Security Considerations
  Allowing direct download via a link in the response might have security implications. It's important to validate user permissions and implement proper security measures.
- Lack of Pagination
  If the client_income_data table contains a large amount of data, exporting all records at once could impact performance. Implementing pagination or a limit on the number of exported records might be beneficial.
- Dependence on External Libraries
  The code relies on external libraries such as fs and fast-csv, which may introduce dependencies and potential issues if not managed properly.
- Debugging Challenges
  The code sends a script to the client for file download, which might make debugging and understanding the flow less straightforward.

4. A recent client partner wanted us to send an SMS to the customer whose details were collected in the response as soon as the ingestion was completed 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.

Middleware Approach:

Create a sample Postgres based Relation database

```
CREATE TABLE client_income_data (
  client_id SERIAL PRIMARY KEY,
  client_email VARCHAR(255),
  client_name VARCHAR(255),
  income_per_annum INT,
  savings_per_annum INT,
  mobile_number VARCHAR(10)
);
```

Route to send SMS using fast-two-sms library

```
// Middleware to send SMS - Task 4
async function SMS(req, res) {
    try {
        const { client_email, client_name, income_per_annum, savings_per_annum,
mobile_number } = req.body;
        var options = {
            authorization: 'Your_API_Key',
            message: ` Your Details :\n Email ID :${client_email}\n Name :
${client_name}\n Income Per Annum: ${income_per_annum}\n Savings Per Annum:
${savings_per_annum}\n Contact : ${mobile_number}\n Thank you for your response`,
            numbers: [mobile_number]
        };

        const response = await fast2sms.sendMessage(options); // Asynchronous
Function.

        console.log('SMS Response:', response);

        if (response) {
            if (response.status === 'OK') {
                res.send(response.message);
```

```
            } else {
                console.error('Error sending SMS:', response);
                res.status(500).send(`Error sending SMS: ${response &&
response.message ? response.message : 'Unknown error'}`);
            }
        } else {
            console.error('Error sending SMS: Unknown error');
            res.status(500).send('Error sending SMS: Unknown error');
        }
    } catch (err) {
        console.error('Error during SMS sending:', err.message);
        res.status(500).send(`Error during SMS sending: ${err.message}`);
    }
}
```

Pros:

- SMS Integration: The server integrates with the fast-two-sms library to send SMS messages, which can be useful for notifications or alerts.

Cons:

- API Key Exposure: The API key for sending SMS is hardcoded in the server file, which is not a secure practice. Consider using environment variables or a secure configuration management approach.