Assignment No: 2

Title: Flow Distribution Algorithm (Backend)

Problem Statement: Design and implement a flow distribution algorithm in Node.js for connecting users with astrologers. The goal is to ensure that each astrologer gets an equal proportion of chances to connect with users, while also providing the flexibility to adjust flow for top astrologers.

Prepared By: Saurabh Kumar

Email: dypu.saurabh11@gmail.com

Phone: +91 8252702699

Submitted to:

Guruji Astro

INDEX

1. INTRODUCTION

1.1 Assignment Overview:

This backend development assignment focuses on designing and implementing a Flow Distribution Algorithm using Node.js. The goal of this system is to intelligently and fairly allocate incoming user queries to a pool of available astrologers. The distribution must ensure an equal and optimized workload among astrologers, while still maintaining the ability to prioritize top-performing astrologers through flexible configuration.

In modern digital consultation platforms, where thousands of users may request services daily, it is crucial to have a backend mechanism that not only handles the scale but also does so intelligently, fairly, and efficiently. The Flow Distribution Algorithm is developed with these core principles in mind.

1.2 Purpose of the Assignment:

The primary purpose of this assignment is to simulate a real-world backend system where users (clients) are automatically assigned to service providers (astrologers) based on dynamic criteria such as:

- **Current availability**
- **Specialization match**
- **Historical load (assignments)**
- **Custom preference for top astrologers**

This system must handle a **daily traffic load of 2000–3000 users** and manage **a pool of 500+ astrologers**, ensuring both efficiency and fairness in routing.

1.3 Key Functional Goals:

1. **Fair Load Distribution**: Distribute user queries evenly among astrologers based on previous assignments and current availability.
2. **Specialization-Based Filtering**: Assign astrologers who match the user's required domain (e.g., Tarot, Palmistry, Vedic).
3. **Dynamic Flow Control**: Enable increased or decreased query flow for "top astrologers" via a toggle or weighting system.
4. **High Scalability**: Ensure the system remains stable and responsive under high-volume usage.
5. **API Accessibility**: Provide RESTful API endpoints for distributing tasks and managing flow logic.
6. **Testability**: Create unit tests to validate functionality and resilience across edge cases.

1.4 Technologies & Tools Used:

- **Node.js** – Runtime environment for server-side code execution.
- **Express.js** – Web framework used to build RESTful APIs.
- **MongoDB + Mongoose** – NoSQL database and object modeling for managing astrologers and assignment data.
- **dotenv & cors** – Environment variable management and cross-origin handling.
- **Jest & Supertest** – Testing libraries for unit and integration tests.
- **Postman** – For manual testing and API documentation.

2. OBJECTIVES

2.1 Goal

The main objective of this project is to develop a scalable and intelligent backend system that distributes user requests to astrologers in a way that is fair, balanced, and adaptable to changing business priorities (e.g., promoting top astrologers). The system is designed to handle high daily user traffic and must ensure optimized routing based on astrologer availability, specialization, and engagement history.

2.2 Specific Objectives

1. **Even Load Distribution**
   Ensure all astrologers receive a fair number of user connections by tracking previous assignments and using round-robin or weighted distribution strategies.
2. **Specialization Matching**
   Route users to astrologers based on their preferred service domain (e.g., Tarot, Palmistry, Numerology), improving user satisfaction by providing domain experts.
3. **Support for Top Astrologer Flow Control**
   Incorporate a toggle mechanism or weight-based system to allow certain "top astrologers" to receive **increased or decreased user flow** based on their popularity or performance metrics.
4. **Scalable and High-Performance Logic**
   Architect the distribution logic to handle large-scale traffic, such as **2000–3000 user requests per day**, without performance bottlenecks.
5. **Flexible Algorithm Design**
   Design the distribution flow in a modular way so it can be easily enhanced in the future with AI-based scoring, priority queues, or availability detection via WebSockets.
6. **Testability and Maintainability**
   Include **unit tests** and **integration tests** using Jest and Supertest to validate the correctness and stability of the system under different conditions, ensuring maintainable codebase.
7. **Seamless API Integration**
   Expose RESTful API endpoints for backend communication, including:
   - Retrieving active astrologers
   - Assigning users
   - Toggling top astrologer preferences
   - Viewing distribution logs

3. WORKING PRINCIPLE

The Flow Distribution Algorithm follows a deterministic yet adaptable sequence of operations to ensure fair and intelligent routing of user requests to astrologers. The primary objective is to balance the load evenly while allowing customization for premium or top-performing astrologers. Below is a step-by-step breakdown of how the system operates:

3.1 Step-by-Step Flow Logic

Step 1: Fetch All Active Astrologers

- Query the MongoDB database for astrologers where:
  - status: "active"
- This ensures that only those astrologers currently available are considered in the distribution.

Step 2: Filter by Specialization (if specified)

- If a user's request specifies a preferred specialization (e.g., Palmistry, Tarot):
  - Filter the astrologer list to include only those whose specialization array includes that domain.
- If no specialization is mentioned, continue with all active astrologers.

Step 3: Sort Astrologers for Fairness

To ensure fair distribution:

- Astrologers are sorted by either:
  - assignedCount (total number of users they've been assigned)
  - or lastAssigned timestamp (who was assigned a user the longest time ago)

Step 4: Apply Top Astrologer Boost (Optional)

- Some astrologers may be marked as "top astrologers" via a boolean field like isTopAstrologer: true
- These astrologers may be assigned:
  - More flow using a weighted logic (e.g., higher weight in selection)
  - Or less flow, depending on business configuration

This helps promote star performers or reduce flow during off-times.

Step 5: Assign the Astrologer

- The first astrologer from the filtered and sorted list is selected.
- User assignment is performed and persisted:
  - Update assignedCount (+1)
  - Update lastAssigned (Date.now())

Step 6: Update System State

- The astrologer record is updated in the database.
- A log of the assignment is maintained for traceability and analytics.

4. DATA MODELS (MONGODB SCHEMAS)

To implement the Flow Distribution Algorithm effectively, well-structured, and optimized MongoDB schemas are essential. Below are the primary collections used in the backend system:

4.1 Astrologer Model

This model stores the core profile and system-specific data of each astrologer. It plays a crucial role in the assignment algorithm. The **Astrologer Model** represents each astrologer's profile and real-time allocation state within the system. This schema is central to the **Flow Distribution Algorithm**, as it holds the attributes required for assigning users effectively and fairly.

4.1.1 Field Breakdown

1. **name: String**
   - Represents the full name of the astrologer.
   - Used for identification and display purposes.
2. **specialization: [String]**
   - An array of specialization areas the astrologer practices.
   - Example: ["Tarot", "Palmistry", "Vedic"]
   - Helps filter astrologers based on user query types.
3. **assignedCount: Number**
   - Default: 0
   - Keeps track of how many users have been assigned to this astrologer.
   - Used to ensure a fair distribution of work and avoid overloading any single astrologer.
4. **lastAssigned: Date**
   - Timestamp of when the astrologer was last assigned to a user.
   - Helps in implementing a **round-robin logic** to evenly spread assignments.
   - Astrologers with older lastAssigned times are prioritized.
5. **isTopAstrologer: Boolean**
   - Default: false
   - Indicates if the astrologer is a **high-performing** or **priority** astrologer.
   - This field allows **manual or algorithmic boosts** in assignment frequency if enabled in the logic.

6. **status: 'active' | 'inactive'**
    - Default: 'active'
    - Represents the availability status of the astrologer.
    - Only astrologers with status = 'active' are considered for new assignments.
    - Can be toggled manually or automatically based on activity.

4.1.2 Usage in Flow Distribution Algorithm

- **Filtering**: First filters by status = active and matching specialization.
- **Sorting**: Sorts filtered astrologers by assignedCount or lastAssigned for fairness.
- **Control**: isTopAstrologer allows control over flow preference or boosting.
- **Scalability**: The model is designed to handle hundreds of astrologers efficiently, allowing the algorithm to function smoothly even at scale (e.g., 500 astrologers).

4.2 User Request Model

The **User Request Model** represents each incoming user query or request that needs to be assigned to an astrologer. This model is vital for tracking user preferences and ensuring that each query is appropriately matched with a compatible astrologer.

4.2.1 Field Breakdown

1. **userId: String**
    - A unique identifier for the user (could be auto-generated or fetched from an authentication system).
    - Helps in tracking user history, requests, and assignments.
2. **queryType: String**
    - Specifies the type of help the user is seeking.
    - Example: "Palmistry", "Tarot", "Numerology", etc.
    - Used to match against astrologer specialization.
3. **timestamp: Date**
    - Automatically set to the current date and time when the request is created.
    - Helps in managing real-time queueing and sorting if multiple requests come in simultaneously.
4. **status: 'pending' | 'assigned' | 'queued' | 'resolved'**
    - Tracks the current state of the user's query in the flow.
    - pending: Awaiting assignment

  - assigned: Assigned to an astrologer
  - queued: Temporarily held (e.g., no astrologer currently available)
  - resolved: Completed by astrologer

5. **assignedAstrologer: String | null**
   - Holds the _id or name of the astrologer once a match is made.
   - Initially null until assigned.
   - Enables tracking which astrologer handled which query.

4.2.2 Usage in Flow Distribution Algorithm

- **Entry Point**: Every new user request is stored using this model.
- **Query Matching**: The algorithm reads queryType and filters astrologers accordingly.
- **State Tracking**: status updates during various stages of the flow—helpful for debugging and dashboard visualization.
- **Assignment**: Once matched, assignedAstrologer is updated and passed for logging or user notification.

4.2.3 Design Considerations

- Designed for high-frequency use (2000–3000 users/day).
- Can be extended to include user feedback, priority, urgency, or language preferences.
- Structured to support both synchronous and asynchronous assignment processes (real-time or queued).

4.3 Why These Fields Matter in the Algorithm

Field Name	Purpose
assignedCount	Used to track the load and balance requests evenly.
lastAssigned	Helps in implementing round-robin distribution.
specialization	Enables intelligent filtering for targeted queries.
isTopAstrologer	Allows fine control over premium astrologers' visibility in the flow.
status	Ensures only available astrologers are considered for distribution.

4.4 Indexing for Performance

To handle high volumes (2000–3000 daily requests):

- Index status, specialization, and lastAssigned fields in the **Astrologer** collection.
- This drastically improves query performance during the assignment step.

5. SAMPLE FLOW SCENARIO

To illustrate the working of the Flow Distribution Algorithm, let's consider a real-world simulation involving a user query and a pool of astrologers.

5.1 Scenario Setup:

- **Total Active Astrologers:** 3
    - Astrologer A: Specializes in Palmistry, assignedCount = 10, lastAssigned = 1 hour ago
    - Astrologer B: Specializes in Tarot, assignedCount = 7, lastAssigned = 45 minutes ago
    - Astrologer C: Specializes in Palmistry and Tarot, assignedCount = 5, lastAssigned = 2 hours ago
- **Incoming User Request:**
    - queryType = "Palmistry"

5.2 Step-by-Step Flow:

1. **Fetch Active Astrologers:**
    - The system queries the database for astrologers with status = "active".
2. **Filter by Specialization:**
    - Only astrologers whose specialization includes **"Palmistry"** are selected.
    - **Filtered List:** Astrologer A and Astrologer C
3. **Sort by Least Assigned:**
    - Among the filtered astrologers:
        - Astrologer A → assignedCount = 10
        - Astrologer C → assignedCount = 5
    - The system selects **Astrologer C** as the best match.
4. **Assignment and Update:**
    - The request is assigned to **Astrologer C**.
    - assignedCount is incremented to 6.
    - lastAssigned is updated to the current timestamp.
    - The user request's status is set to **"assigned"** and assignedAstrologer is updated accordingly.

5.3 Result:

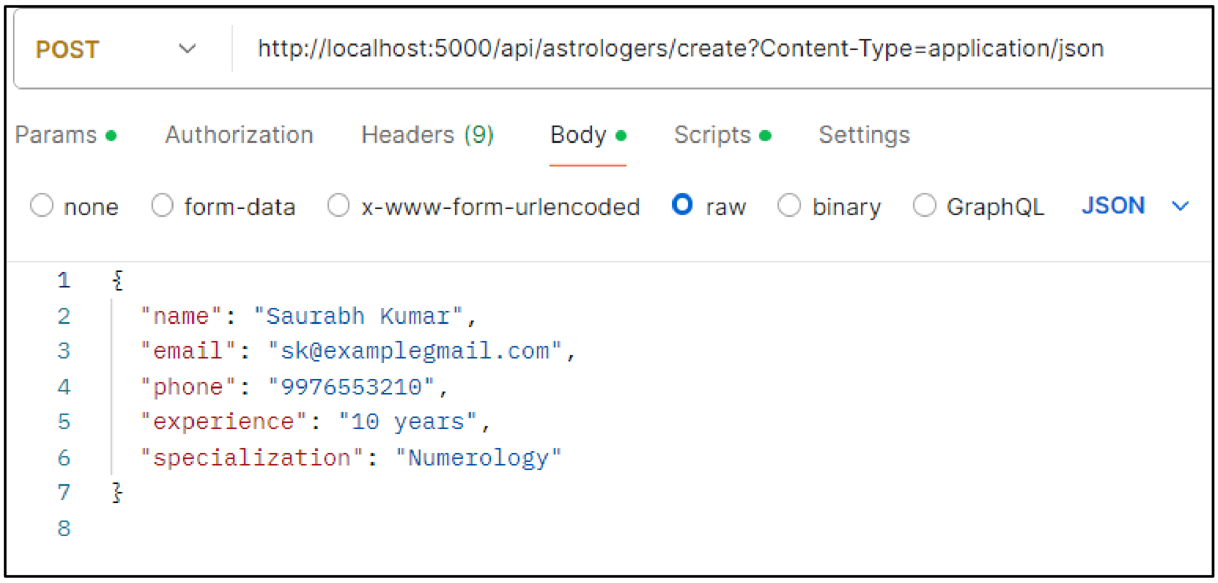
- **Astrologer C** receives the request.
- The algorithm ensures fair load balancing by preferring the astrologer with the **least workload and oldest assignment** time.
- The flow remains dynamic and adapts as more astrologers or requests are added.

6. API USAGE & INTEGRATION

The Flow Distribution Algorithm provides a set of RESTful API endpoints that enable seamless interaction with the backend system. Below is a breakdown of the available methods and their descriptions:

6.1. POST /api/astrologers/add

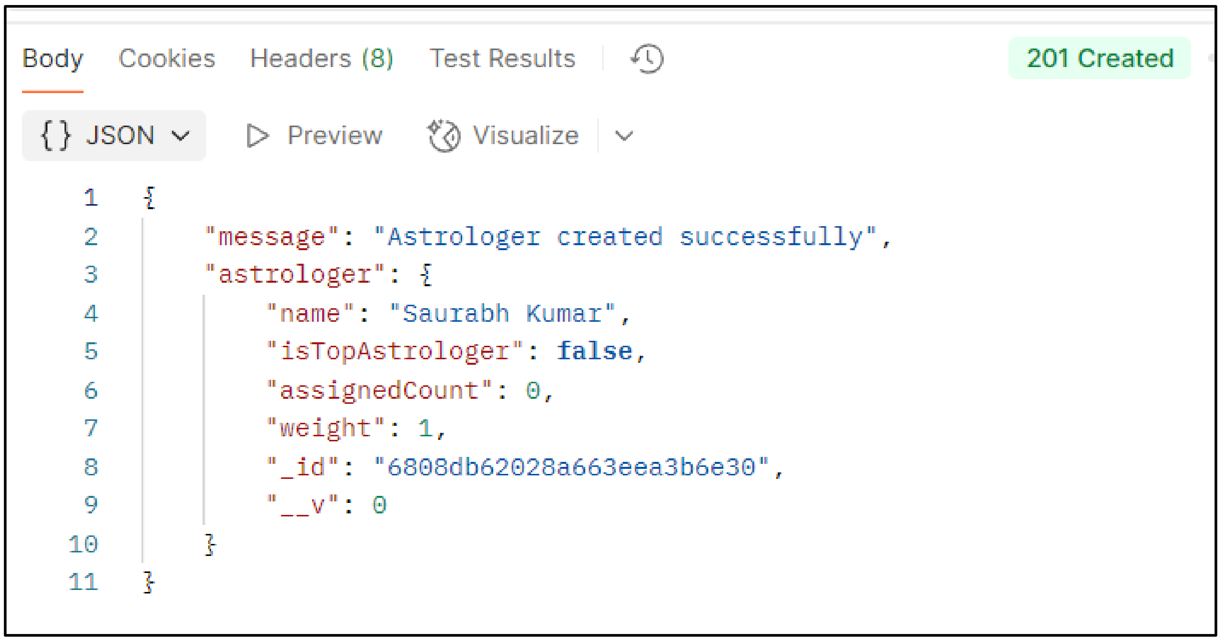
- **Description:** This endpoint allows the addition of new astrologers to the system. Administrators can provide the astrologer's name, email, phone, experience and specialization(s).
- **Request Body:**



```
http://localhost:5000/api/astrologers/create?Content-Type=application/json
```

```
{
  "name": "Saurabh Kumar",
  "email": "sk@examplegmail.com",
  "phone": "9976553210",
  "experience": "10 years",
  "specialization": "Numerology"
}
```

- **Response:**

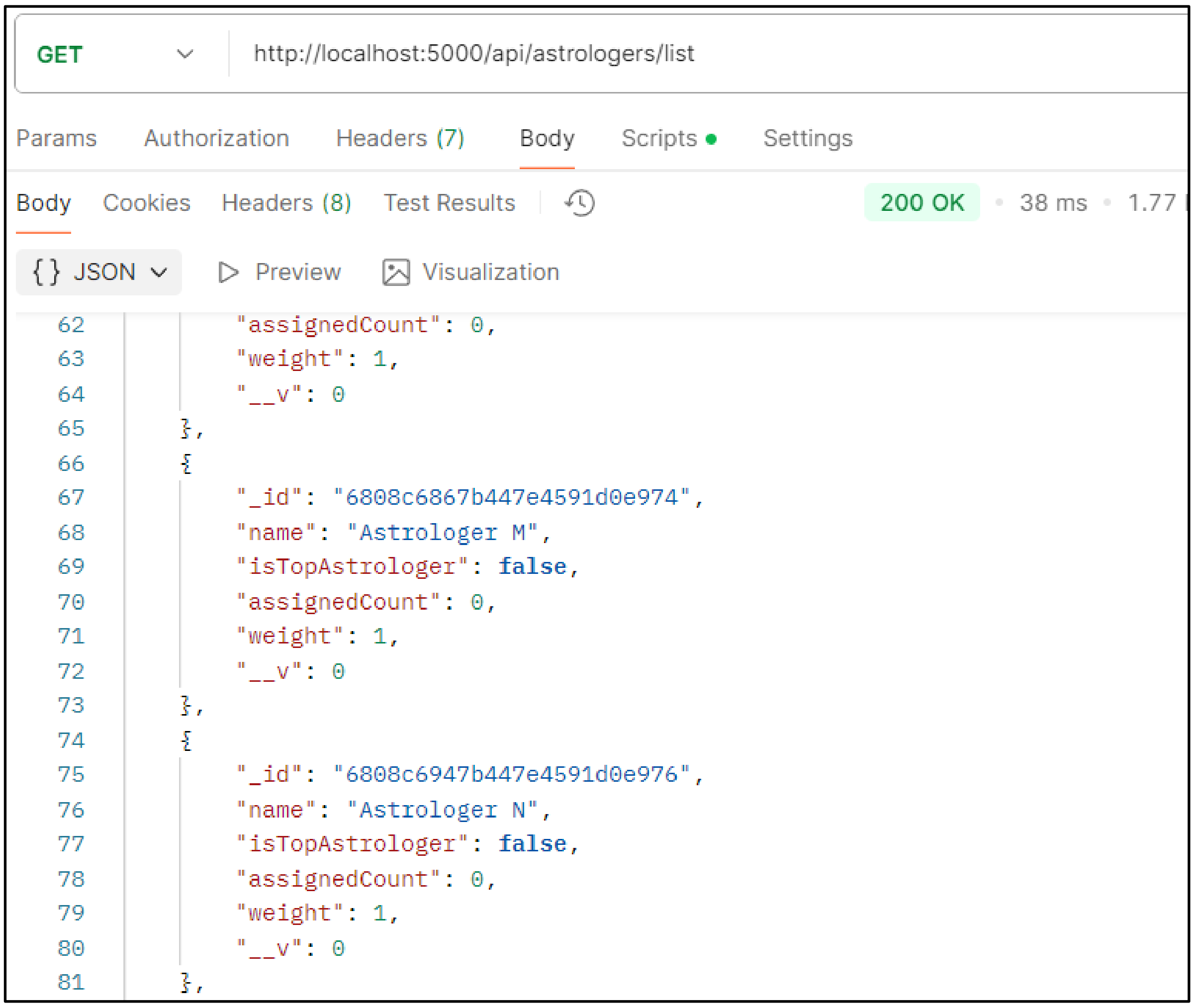


```
{
    "message": "Astrologer created successfully",
    "astrologer": {
        "name": "Saurabh Kumar",
        "isTopAstrologer": false,
        "assignedCount": 0,
        "weight": 1,
        "_id": "6808db62028a663eea3b6e30",
        "__v": 0
    }
}
```

6.2 GET/api/astrologers

- **Description:** This endpoint retrieves a list of all astrologers present in the system. It can be used for administrative purposes or for displaying available astrologers in the user interface.

- **Response:**

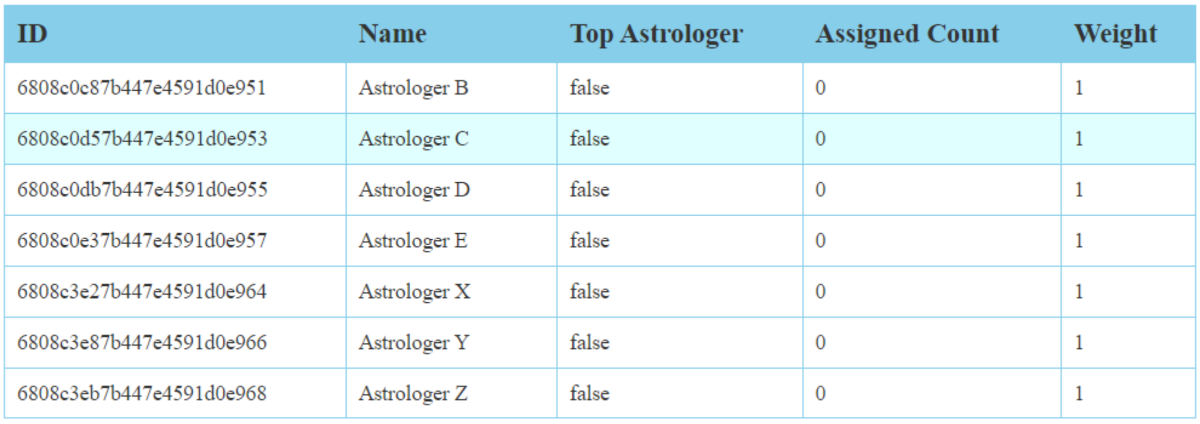


```
GET   http://localhost:5000/api/astrologers/list
```

```
200 OK • 38 ms • 1.77
```

```
        "assignedCount": 0,
        "weight": 1,
        "__v": 0
    },
    {
        "_id": "6808c6867b447e4591d0e974",
        "name": "Astrologer M",
        "isTopAstrologer": false,
        "assignedCount": 0,
        "weight": 1,
        "__v": 0
    },
    {
        "_id": "6808c6947b447e4591d0e976",
        "name": "Astrologer N",
        "isTopAstrologer": false,
        "assignedCount": 0,
        "weight": 1,
        "__v": 0
    },
```

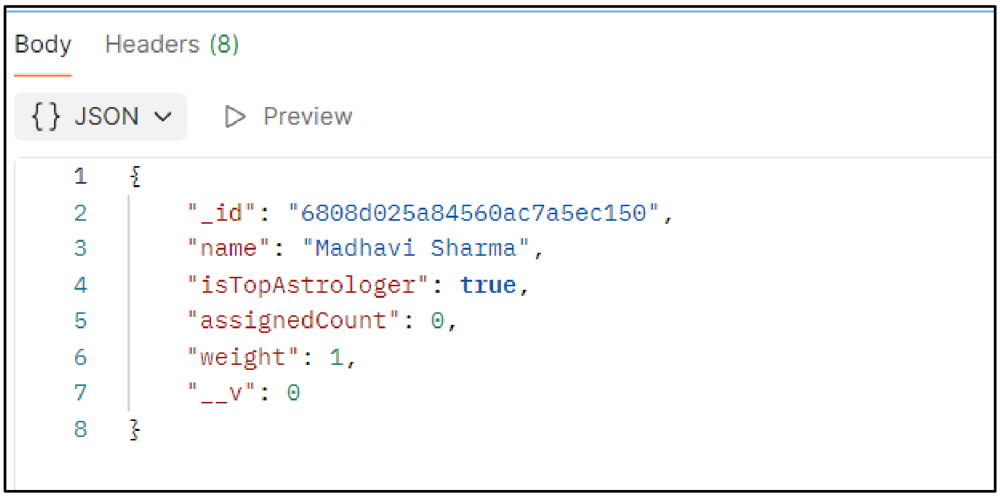
- **Visualization**



ID	Name	Top Astrologer	Assigned Count	Weight
6808c0c87b447e4591d0e951	Astrologer B	false	0	1
6808c0d57b447e4591d0e953	Astrologer C	false	0	1
6808c0db7b447e4591d0e955	Astrologer D	false	0	1
6808c0e37b447e4591d0e957	Astrologer E	false	0	1
6808c3e27b447e4591d0e964	Astrologer X	false	0	1
6808c3e87b447e4591d0e966	Astrologer Y	false	0	1
6808c3eb7b447e4591d0e968	Astrologer Z	false	0	1

6.3 **PATCH /api/astrologers/:id/toggle-flow**

- **Description:** This endpoint is used to toggle the flow mode for a specific astrologer. When an astrologer is designated as a "top astrologer," they will receive more user requests than others, based on the predefined configuration. This endpoint allows enabling or disabling the top astrologer mode for a specific astrologer by changing their flow preferences.
- **Parameters:**
  - :id: The unique ID of the astrologer whose flow preference is to be toggled.
- **Response**



```
{
    "_id": "6808d025a84560ac7a5ec150",
    "name": "Madhavi Sharma",
    "isTopAstrologer": true,
    "assignedCount": 0,
    "weight": 1,
    "__v": 0
}
```

Conclusion: These API endpoints provide full access to the backend system, allowing for seamless integration with front-end systems or any other service that requires interaction with the astrologer assignment process.

7. TESTING STRATEGY

To ensure the reliability, performance, and correctness of the Flow Distribution Algorithm, we adopt a comprehensive testing strategy that includes **unit tests**, **stress testing**, and **mock database setup**.

7.1 Unit Tests:

Unit tests are written to validate the core functionalities and ensure that the system behaves as expected in various scenarios. These tests focus on the following key areas:

1. **Selection Logic (Round-Robin & Specialization Filtering):**
   - **Objective:** Ensure that the algorithm is correctly selecting the best astrologer based on the user's query and available astrologers.
   - **Test Cases:**
     - **Round-Robin Distribution:** Verify that users are assigned to astrologers in a round-robin fashion, ensuring that astrologers are given equal chances to handle users over time.
     - **Specialization Filtering:** Ensure that the astrologer is selected based on the user's specified specialization (e.g., Tarot, Palmistry) and that the selection respects the specialization requirement.
2. **Priority Toggle Effects:**
   - **Objective:** Verify that the system correctly adjusts the flow distribution when an astrologer is toggled into "top astrologer" mode.
   - **Test Cases:**
     - **Top Astrologer Flow:** Ensure that astrologers designated as top astrologers receive a higher number of requests compared to others.
     - **Toggle Functionality:** Verify that the toggle flow correctly switches between top astrologer mode and regular mode without affecting the overall system stability.

7.2 Mock DB Setup for Jest using In-memory MongoDB:

To isolate unit tests and ensure consistent behaviour without depending on a live database, we use **mongodb-memory-server**, which provides an in-memory MongoDB instance for testing purposes.

1. **Setup:**
   - **Objective:** Set up an in-memory MongoDB instance for testing with **Jest** and **supertest**.
   - **Test Cases:**

    - Ensure that astrologer data can be added, updated, and retrieved correctly from the mock database.
    - Test edge cases like when no astrologers are available or when all astrologers are marked as inactive.

2. **Mock Database Operations:**
    - Using **mongodb-memory-server**, simulate database interactions to test the API routes without interacting with the actual database.
    - This ensures that the tests are fast, isolated, and don't affect the actual data stored in the live database.

8. SCALABILITY & PERFORMANCE OPTIMIZATIONS

Ensuring that the Flow Distribution Algorithm performs efficiently under heavy user load is critical for the success of the system. In this section, we outline several strategies and techniques to improve scalability, minimize latency, and enhance overall performance.

8.1 In-memory Cache for Astrologer Data (Redis or Node Cache):

To reduce the load on the database and improve the response time for assigning users to astrologers, we can leverage an in-memory cache like **Redis** or **Node Cache**. Caching frequently accessed data, such as active astrologer information, can significantly reduce database query time and improve system performance.

1. **Objective:**
    - Store astrologer data, including their availability, specialization, and assigned counts, in a fast-access in-memory cache.
    - When a new request comes in, the system can first check the cache before querying the database, resulting in faster response times.
2. **Implementation:**
    - Use **Redis** or **Node Cache** to store data in memory.
    - Cache astrologer information that doesn't change frequently (e.g., specialization, status, assigned count).
    - Set cache expiration for astrologer data (e.g., every 10 minutes) to avoid serving outdated data.
    - Cache can also be used to store temporary results for complex queries, such as the most recent active astrologers or the most frequently requested specializations.
3. **Benefits:**
    - **Reduced Latency:** By avoiding frequent database hits, we can drastically reduce the response time for querying astrologer information.
    - **Better Scalability:** Handling a large number of concurrent requests becomes much more feasible as database load is significantly reduced.

8.2 Use Weighted Random Sampling for Dynamic Top Astrologer Distribution:

To allow for the dynamic adjustment of the flow towards top astrologers, a **weighted random sampling** approach can be used. This ensures that top astrologers receive more assignments based on their popularity or the flow toggle, while also maintaining fairness for others.

1. **Objective:**
    - Allow astrologers to be assigned a user request based on their weight (higher weight for top astrologers).

    - Adjust the weight dynamically, so that astrologers can be favored or deprived of assignments based on their current status or specializations.

2. **Implementation:**
    - **Assign Weights:** Every astrologer can be assigned a weight value. For example, a top astrologer may have a higher weight (e.g., 5), while others have a standard weight (e.g., 1).
    - **Weighted Sampling:** When a request comes in, the algorithm uses **weighted random sampling** to choose an astrologer. The higher the weight, the more likely the astrologer will be selected.
    - **Adjusting Weights Dynamically:** The weight of an astrologer can be adjusted manually through an admin panel or automatically based on the number of requests they've handled or their rating.
3. **Benefits:**
    - **Fair Distribution:** Top astrologers will be assigned more users as required, but others are still given a fair chance to receive assignments.
    - **Flexibility:** It allows for easy adjustment of the flow distribution without needing to change the core algorithm.

8.3 Batched Distribution (Bulk Insert & Update Instead of One by One):

When multiple user requests come in, performing database operations like inserting or updating records for each user individually can lead to significant performance bottlenecks. To optimize this, **batched operations** should be used to insert or update records in bulk.

1. **Objective:**
    - Minimize the number of database operations required for each request by grouping multiple actions (e.g., assigning users, updating astrologer counts) into one batch operation.
2. **Implementation:**
    - Use MongoDB's **bulkWrite()** method to perform multiple insert or update operations in a single request.
    - Group user assignments, updates to astrologer counts (e.g., incrementing the assignedCount), and any other changes (e.g., updating lastAssigned timestamps) into one batch operation.
    - Ensure that the batch size is kept within reasonable limits to avoid excessive memory usage.
3. **Benefits:**
    - **Reduced Latency:** By sending fewer database operations, the overall system can handle higher throughput.
    - **Efficiency:** Grouping operations reduces the overhead of making multiple round-trip requests to the database.

8.4 Index MongoDB on Specialization, AssignedCount for Faster Lookups:

MongoDB provides the ability to create **indexes** on frequently queried fields. In this case, creating indexes on fields such as **specialization** and **assignedCount** can greatly speed up query execution, especially when the system has to handle large numbers of astrologers and users.

1. **Objective:**
   - Improve query performance by indexing fields that are frequently used in search operations, such as filtering by specialization or sorting by assigned count.
2. **Implementation:**
   - Create indexes on the following fields in the **Astrologer model**:
     - **specialization**: Indexing this field will allow the system to quickly find astrologers with the required specialization (e.g., Tarot, Palmistry).
     - **assignedCount**: Indexing this field will speed up the selection process when assigning users based on the least recently assigned astrologer or balancing the load.

```
astrologerSchema.index({ specialization: 1 });
astrologerSchema.index({ assignedCount: 1 });
```

3. **Benefits:**
   - **Faster Query Execution:** Queries that involve searching by specialization or sorting by assignedCount will be executed much faster, especially when dealing with large datasets.
   - **Scalability:** As the number of astrologers and users grows, indexed queries will continue to perform efficiently, helping the system scale to meet demand.

9. SECURITY MEASURES

Ensuring the security of the application is critical, especially when dealing with sensitive user data and interactions. The following measures have been incorporated to enhance the security of the system:

9.1 Use JWT or API Keys to Secure Endpoints:

To secure the API endpoints and ensure that only authenticated users can access certain functionalities, **JWT (JSON Web Tokens)** or **API Keys** should be implemented.

1. **Objective:**
   - JWT ensures secure and stateless authentication between the client and server.
   - API keys can also be used for certain types of requests that require restricted access.
2. **Implementation:**
   - When a user logs in or registers, they receive a JWT or API key that must be included in the header of subsequent API requests.
   - **JWT**: The token contains user-specific information, and once validated, allows users to access restricted resources.
   - **API Key**: Unique keys assigned to clients or services, used for authenticating and validating their access to the system.
3. **Benefits:**
   - **Stateless Authentication:** No need to store session data on the server.
   - **Access Control:** Restricts access to sensitive data and operations, ensuring that only authorized users can perform certain actions.

9.2 Input Validation Using express-validator:

To prevent invalid data from entering the system and causing potential issues or security vulnerabilities, **input validation** is essential. Using libraries like **express-validator**, we can ensure that the data passed to the system is correctly formatted and valid.

1. **Objective:**
   - Validate user input to ensure that all data is of the expected type, format, and within acceptable ranges.
2. **Implementation:**
   - Use **express-validator** middleware to validate and sanitize incoming data.
   - Validate fields such as email, phone numbers, and user inputs to prevent incorrect or malicious data.

```
const { body } = require('express-validator');

app.post('/api/astrologer', [

body('email').isEmail().withMessage('Invalid email address'),

body('specialization').isIn(['Tarot', 'Palmistry', 'Vedic']).withMessage('Invalid specialization')

]);
```

3. **Benefits:**
   - **Data Integrity:** Ensures that only valid and properly formatted data is accepted.
   - **Prevention of Malicious Data:** Helps mitigate attacks like SQL injection or other forms of data manipulation.

9.3 Sanitize Incoming Data to Prevent Injection Attacks:

Data sanitization ensures that any potentially harmful or executable code within the input data is removed before being processed by the system. This step helps protect the system from **injection attacks** such as SQL injections or cross-site scripting (XSS).

1. **Objective:**
   - Remove any malicious scripts or characters from user inputs that could cause harm to the system or database.
2. **Implementation:**
   - Use sanitization libraries like **express-sanitizer** or manually remove potentially harmful characters from user input (e.g., angle brackets, SQL keywords).
   - Ensure that no user input is executed as code or query.
3. **Benefits:**
   - **Prevention of Injection Attacks:** Mitigates the risk of harmful code execution.
   - **Safe Data Handling:** Ensures that only safe, clean data is processed by the backend.

9.4 .env File Secured with Proper Environment Variables for Mongo URI, etc.:

Storing sensitive information such as MongoDB connection strings, JWT secret keys, and other configuration variables should never be hard-coded into the source code. Instead, use an **.env file** to manage environment variables securely.

1. **Objective:**
   - Store sensitive information like MongoDB URI, JWT secret, and other private credentials in an environment file to keep them safe.

2. **Implementation:**
    - Use **dotenv** package to load environment variables from the .env file into your application at runtime.
    - Ensure that the .env file is **not** committed to version control (e.g., Git) to avoid exposing sensitive data.

    ```
    MONGO_URI=mongodb://username:password@hostname:port/dbname

    JWT_SECRET=secret-key
    ```
3. **Benefits:**
    - **Secure Configuration Management:** Keeps sensitive information out of the codebase.
    - **Flexible Environment Management:** Easily switch between different environments (e.g., development, production).

CONCLUSION

In this assignment, I designed and implemented a robust **Flow Distribution Algorithm** in **Node.js** aimed at effectively connecting users with astrologers. The core objective of the system was to ensure an **equal distribution of user requests** among astrologers, while also offering flexibility to prioritize certain astrologers based on their **performance** or **specialization**.

Through the implementation of the **round-robin approach** and **specialization-based filtering**, we achieved a balanced flow distribution, ensuring that every astrologer had an equal opportunity to handle user requests. This was enhanced by a priority toggle mechanism, which allowed for dynamic adjustments to give more exposure to **top astrologers**.

The flow distribution algorithm was designed to handle edge cases and failure scenarios, such as overloaded astrologers, inactive astrologers, or system failures. By integrating features like queuing and retry mechanisms, we ensured that the system remained **resilient** under high demand and provided **fallback solutions** where needed.

Moreover, **scalability** was a key consideration, with optimizations such as in-memory caching, **weighted random sampling** for astrologer assignment, and **MongoDB indexing** for faster lookups. These measures will enable the system to handle future growth and large volumes of requests with ease.

Finally, **security** was prioritized, implementing **JWT authentication**, input validation, and data sanitization to protect sensitive user and astrologer information. This provides a secure and reliable platform for both users and astrologers to interact safely.