Fitpage Coding Assignment

Lathireddi Ekalavya

[ekalavyalatchireddi123@gmail.com](mailto:ekalavyalatchireddi123@gmail.com)

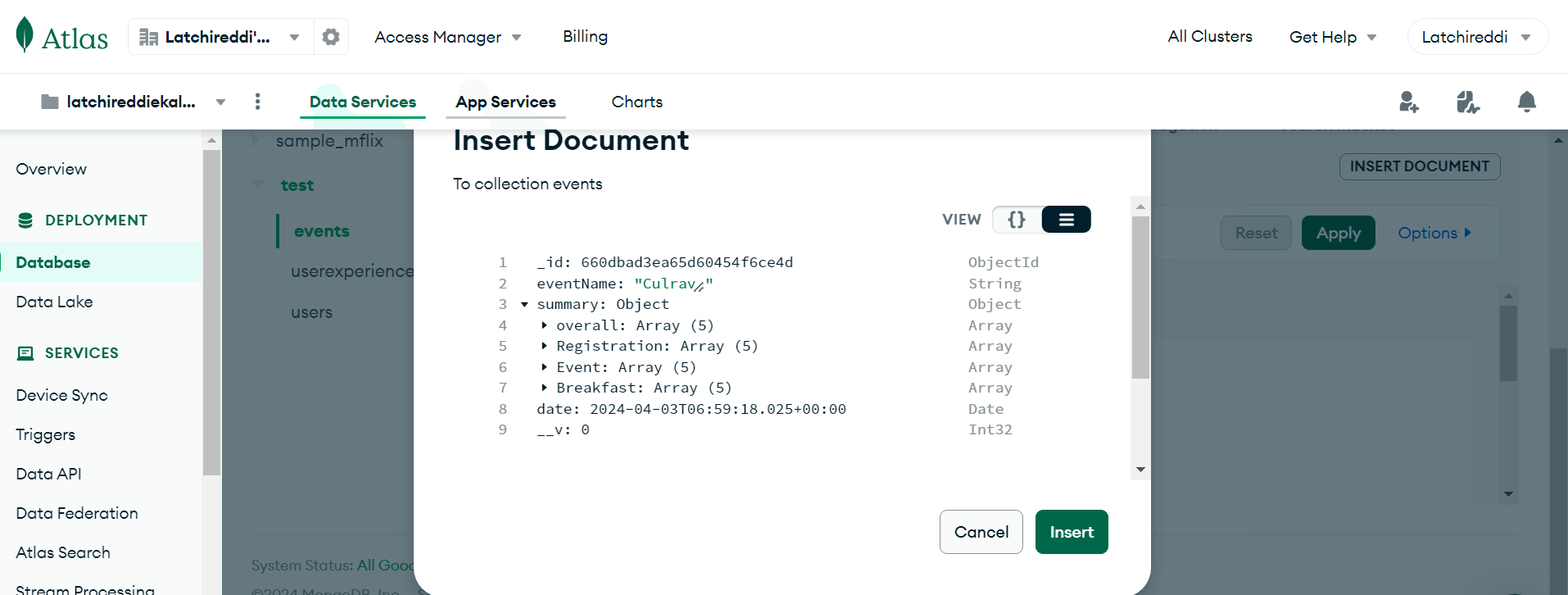
Introduction

This **Node.js** backend project is designed to provide a robust event review system for managing data and user authentication, including email verification. It utilizes **Express.js** for handling HTTP requests and **MongoDB** as the database for data storage. The APIs have been extensively tested using **Postman** to ensure functionality and reliability.

The authentication system includes features such as password hashing using **bcrypt** for enhanced security**, email verification** to ensure valid user accounts, and **JWT tokens** for secure user **authentication** and authorization. These components collectively contribute to a stable and secure backend infrastructure for web applications.

Database Schemas Description

1. **Event Schema**:
   * **Description**: Defines the structure for storing event information.
   * **Fields**:
     + **eventName**: Stores the name of the event (String, required).
     + **summary**: Contains summary data for the event, including overall ratings and specific ratings for registration, event experience, and breakfast (Arrays with default values).
     + **date**: Records the date when the event information is added (Date, default: current date).



**Event Schema Example**

1. **User Schema**:
   * **Description**: Defines the structure for storing user information.
   * **Fields**:
     + **name**: Stores the user's name (String, required).
     + **email**: Stores the user's email address (String, required).
     + **password**: Stores the user's password (String, required).
     + **date**: Records the date when the user account is created (Date, default: current date).
     + **verified**: Indicates whether the user's email is verified (Boolean, default: false).
     + **manager**: Indicates whether the user has manager privileges (Boolean, default: false).



**User Schema Example**

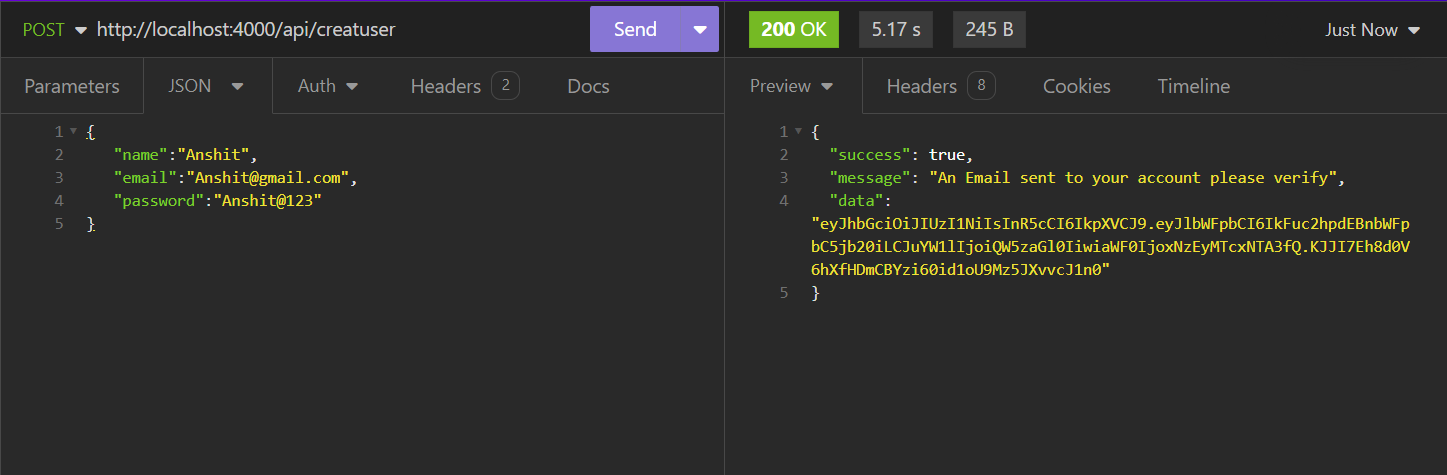
1. **User Experience Schema**:
   * **Description**: Defines the structure for storing user experience related to events.
   * **Fields**:
     + **eventName**: Stores the name of the event related to the user experience (String, required).
     + **email**: Stores the user's email address (String, required).
     + **review**: Contains details of the user's review for the event, including registration, event experience, and breakfast (String with default values).
     + **rating**: Stores numerical ratings for registration, event experience, and breakfast (Numbers, required).
     + **overall**: Stores the overall rating given by the user (Number, required).
     + **likes**: Tracks the number of likes received for the user's review (Number, default: 0).
     + **reports**: Tracks the number of reports received for the user's review (Number, default: 0).
     + **reply**: Stores any reply added by the user or admin (String, default: empty).
     + **flag**: Indicates whether the review has been flagged (Boolean, default: false).
     + **date**: Records the date when the user experience data is added (Date, default: current date).



**User Review Schema Example**

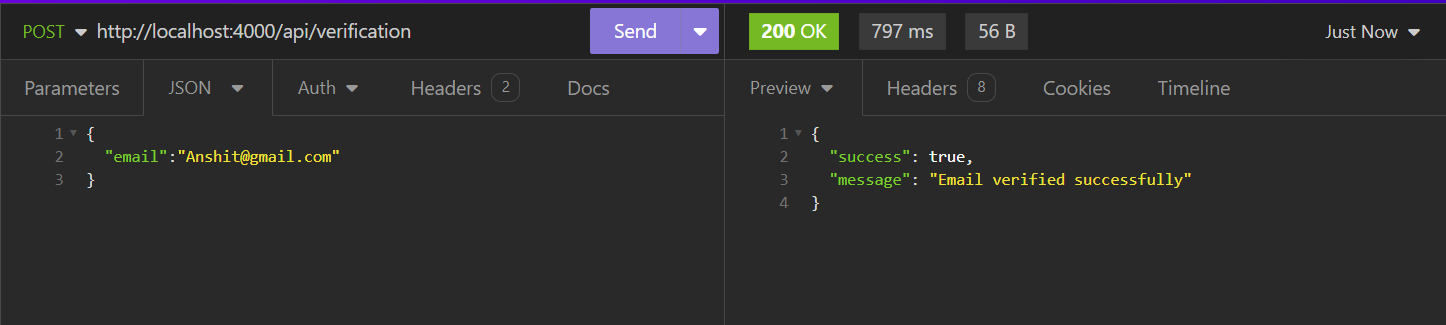
Authentication (API End Points):

1. **POST /creatuser:**
   * Description: Creates a new user account with email, name, and password after checking if the email already exists.
   * Initial Check: Verifies if the provided email is not already registered. If it is, the API responds with a message indicating that the email is already in use.
   * Purpose: Allows users to register and create an account in the system while ensuring uniqueness of email addresses.
   * Request Body: Includes email, name, and password.
   * Response: Indicates success or failure of user creation, including messages for duplicate email or successful registration.

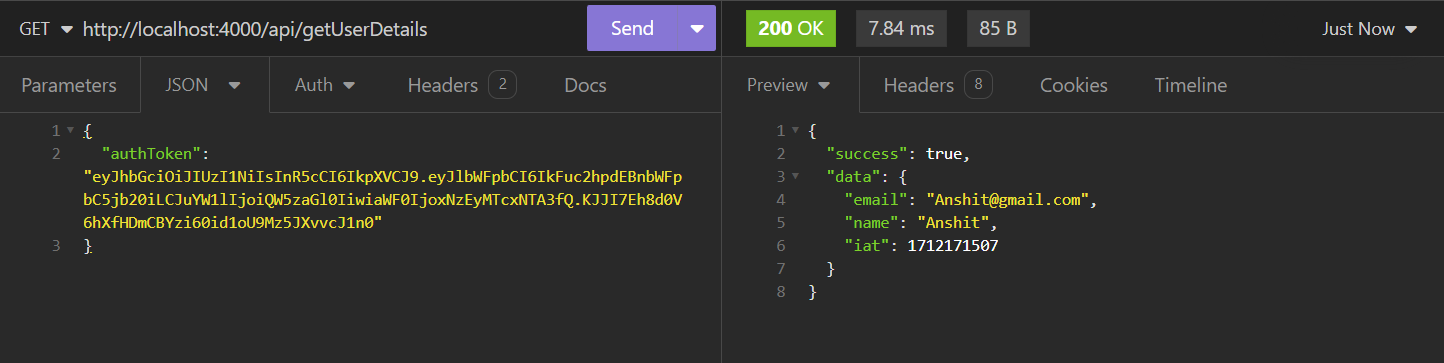


**It sends a email to user email with a token consist user data to verify**

1. **GET /loginuser/:email/:password:**
   * Description: Logs in a user with the provided email and password after validating the email format and password length.
   * Initial Check: Validates the email verified or not , if not verified then an email will be sent to user to verify.
   * Purpose: Authenticates users and generates a JWT token for authorized access.
   * Parameters: User email and password.
   * Response: Returns a JWT token upon successful login or indicates login failures.
2. **POST /verification:**
   * Description: Verifies the user's email address after registration.
   * Purpose: Marks the user as verified to access authenticated functionalities.
   * Request Body: Contains the user's email.
   * Response: Indicates success or failure of email verification.

**when user clicks on link that sent to email , this api call will be called**

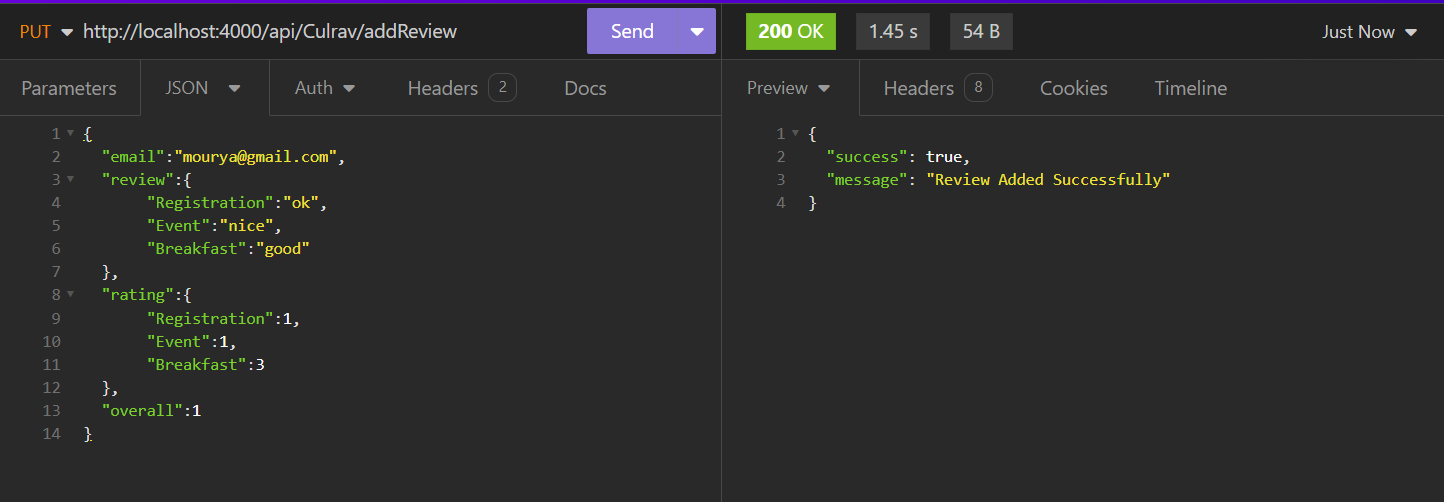
1. POST /forgotPassword:
   * Description: Initiates the process to reset the user's password after checking if the email exists.
   * Initial Check: Verifies if the provided email is registered. If not, the API responds with a message indicating that the email does not exist.
   * Purpose: Provides a way for users to reset their forgotten passwords.
   * Request Body: Includes the user's email for password reset.
   * Response: Sends a password reset email if the email exists in the system.
2. **POST /updatePassword:**
   * Description: Updates the user's password after a password reset request.
   * Purpose: Allows users to set a new password after resetting the old one.
   * Request Body: Contains the user's email and new password.
   * Response: Indicates success or failure of password update after reset.
3. **POST /checkToken:**
   * Description: Validates the JWT token sent by the client.
   * Purpose: Ensures the authenticity and validity of JWT tokens for authorized actions.
   * Request Body: Contains the JWT token to be checked.
   * Response: Indicates whether the token is valid and associated with a user.
4. **GET /getUserDetails**:
   * Description: Retrieves user details based on the JWT token after validating its authenticity.
   * Purpose: Allows users to fetch their profile information securely.
   * Request Body: Contains the JWT token for authentication.
   * Response: Returns user details if the token is valid and associated with a user.



**Gives the user details from AuthToken**

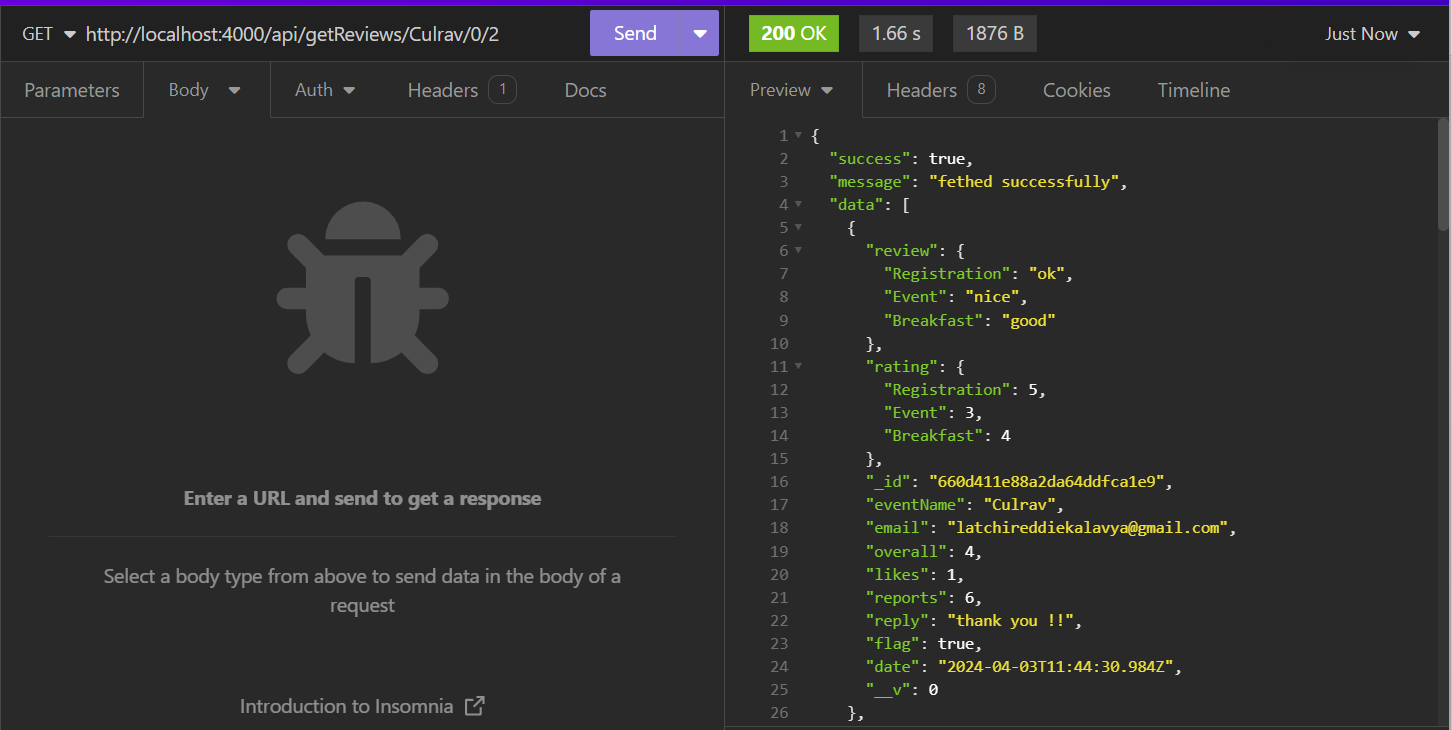
Event Handler (API End Points)

1. **POST /createEvent:**
   * Description: Creates a new event after checking if it already exists.
   * Initial Check: Verifies if the event with the provided name already exists in the database.
   * Purpose: Allows the addition of new events to the system.
   * Request Body: Includes the event name.
   * Response: Indicates success or failure of event creation.
2. **PUT /:event/addReview:**
   * Description: Adds a review to a specific event after validating the email and event existence.
   * Initial Check: Validates the email format and checks if the event exists in the database.
   * Purpose: Enables users to submit reviews for events.
   * Request Body: Contains email, review details, ratings, and overall experience.
   * Response: Indicates success or failure of review submission.



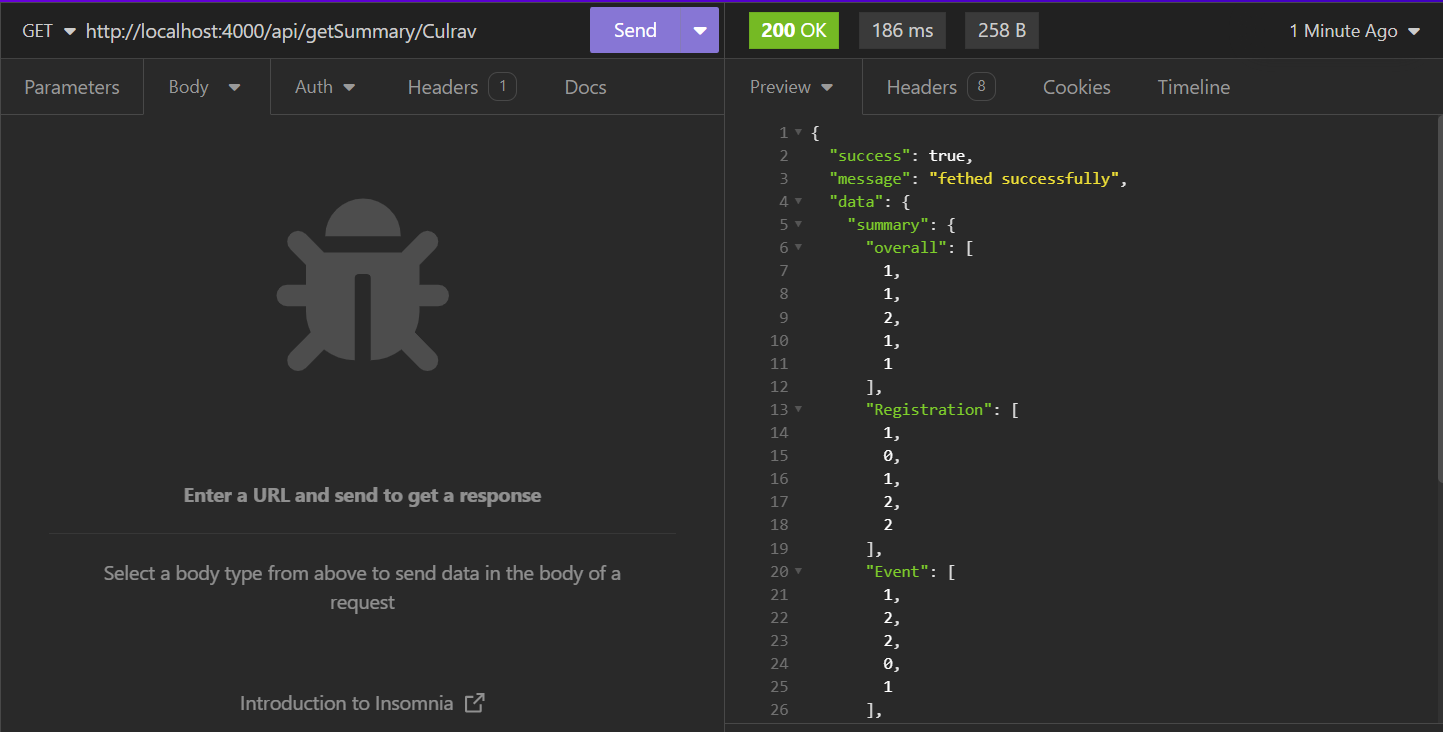
**Review will be added or updates review if review already exist by that user**

1. **GET /getReviews/:event/:startPage/:endPage:**
   * Description: Retrieves reviews for a particular event within a specified range of pages.
   * Initial Check: Verifies if the event exists in the database.
   * Purpose: Facilitates the retrieval of reviews for event analysis.
   * Parameters: Event name, start page, and end page for pagination.
   * Response: Returns reviews data within the specified page range.



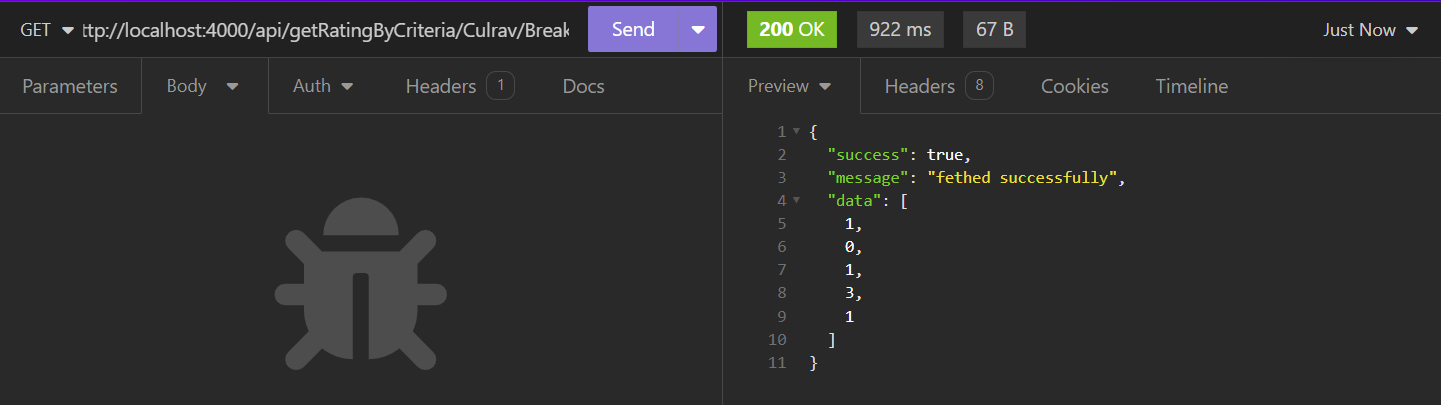
**Retrieve the reviews of specific event for the given range of pages**

1. **GET /getSummary/:event:**
   * Description: Fetches the summary data for a specific event.
   * Initial Check: Checks if the event exists in the database.
   * Purpose: Provides summarized data about an event.
   * Parameters: Event name for which summary is requested.
   * Response: Returns summarized data for the specified event.



**Each category summary consist array of length 5 represents each stars count (like how many 1 star , 2 star , …. , 5 star came for that particular event**

1. **GET /getRatingByCriteria/:event/:criteria:**
   * Description: Retrieves ratings based on a specific criteria for a given event.
   * Initial Check: Validates the existence of the event in the database.
   * Purpose: Allows analysis of ratings based on different criteria for an event.
   * Parameters: Event name and criteria for which ratings are requested.
   * Response: Returns ratings data for the specified criteria of the event.



**For an event particular category summary , returns the array of 5 elements represents the total count of 1 start , 2 star , … , 5 star given by users**

1. **POST /:event/addReply:**
   * Description: Adds a reply to a user's review for a particular event.
   * Initial Check: Validates the email format and checks if the event exists.
   * Purpose: Enables event organizers or managers to respond to user reviews.
   * Request Body: Contains email, event name, and reply content.
   * Response: Indicates success or failure of reply addition.
2. **PUT /:event/addLike:**
   * Description: Increases the like count for a specific event's review.
   * Initial Check: Validates the email format and checks event existence.
   * Purpose: Allows users to like reviews for events they find helpful or enjoyable.
   * Request Body: Contains email and event name.
   * Response: Indicates success or failure of like addition.
3. **PUT /:event/report:**
   * Description: Reports an event's review, potentially flagging it for further review.
   * Initial Check: Validates the email format and verifies event existence.
   * Purpose: Enables users to report reviews that violate guidelines or are inappropriate.
   * Request Body: Contains email and event name for reporting.
   * Response: Indicates success or failure of the report submission.
   * **Note** : if reports become greater that 5 then flag tag will be activated to that review

**Conclusion**

The codebase exhibits a well-structured and modular design, adhering to established coding standards, and is readily comprehensible for other developers. This structure is evident in the organization of the code into distinct modules, such as models (including 'event,' 'user,' and 'userExperience'), routes ('auth' and 'event handler'), and essential files like 'index.js,' '.env,' 'db' (for database connectivity), and 'sendEmail' (for email functionality).

Following consistent naming conventions, such as PascalCase for models and camelCase for variables and functions, contributes to maintaining coding standards throughout the codebase. Each file serves a specific function, contributing to the overall functionality of the application. For example, the 'auth' route handles authentication-related endpoints, while the 'event handler' route manages event-related functionalities.

This modular approach enhances the codebase's readability and maintainability, making it easier for developers to understand and work on different parts of the application independently. It also facilitates scalability, allowing for the seamless addition of new features and modifications to existing functionalities without compromising code integrity or readability.

In summary, the codebase's organization, adherence to coding standards, and focus on modularity ensure that it is not only understandable but also scalable and conducive to collaborative development efforts.