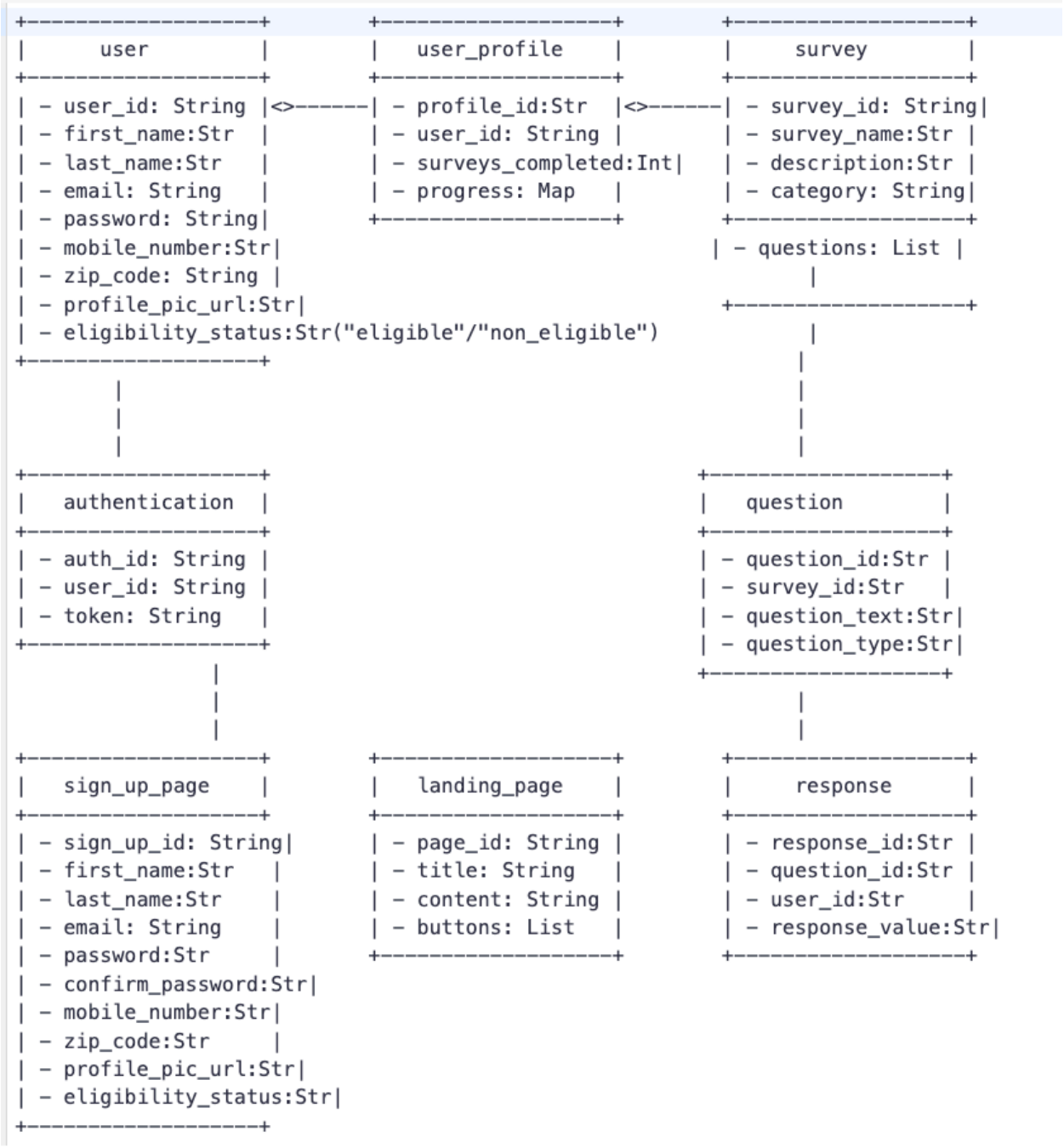
UML Diagram for the User Facing App

This updated UML diagram provides a comprehensive view of the user-facing application, integrating all the components- this includes Sign up, login, auth, Survey, responses and question - system based on the events requirements shared earlier.



Explanation of Classes and Relationships

1. User

- Represents a registered user in the system.
- Attributes:
  - `user_id`: Unique identifier for the user.
  - `first_name`, `last_name`: Personal details.
  - `email`: Email address used for login.
  - `password`: Encrypted password for authentication.
  - `mobile_number`: Phone number for communication.
  - `zip_code`: ZIP code for regional eligibility checks.
  - `profile_pic_url`: URL of the uploaded profile picture.
  - `eligibility_status`: Indicates whether the user is eligible ("eligible" or "non_eligible").
    - Indicates whether the user is eligible to receive event notifications - they can still participate in survey completion.

2. UserProfile

- Represents the user's profile page.
- Attributes:
  - `profile_id`: Unique identifier for the profile.
  - `user_id`: Links the profile to the user.
  - `surveys_completed`: Tracks the number of completed surveys.
  - `progress`: A map that tracks the user's progress in each survey (e.g., `"survey_1": "completed"`).

3. Survey

- Represents a survey with multiple questions.
- Attributes:
  - `survey_id`: Unique identifier for the survey.
  - `survey_name`: Name of the survey.
  - `description`: Description of the survey.
  - `category`: Category of the survey (e.g., "Basics", "Relationship").
  - `questions`: List of questions associated with the survey.

4. Question

- Represents a single question within a survey.
- Attributes:

- `question_id`: Unique identifier for the question.
- `survey_id`: Links the question to its parent survey.
- `question_text`: The text of the question (e.g., "What is your favorite color?").
- `question_type`: Type of question (e.g., `text_input`, `multiple_choice`, `rating_scale`).

5. Response

- Represents a user's response to a specific question.
- Attributes:
  - `response_id`: Unique identifier for the response.
  - `question_id`: Links the response to the question.
  - `user_id`: Links the response to the user who provided it.
  - `response_value`: The value of the response (e.g., "Blue" for a multiple-choice question).

6. Authentication

- Handles user authentication processes.
- Attributes:
  - `auth_id`: Unique identifier for the authentication record.
  - `user_id`: Links the authentication record to the user.
  - `token`: Session token for secure access.

7. SignUpPage

- Represents the sign-up form.
- Attributes:
  - `sign_up_id`: Unique identifier for the sign-up process.
  - `first_name`, `last_name`, `email`, `password`, `confirm_password`, `mobile_number`, `zip_code`, `profile_pic_url`, `eligibility_status`.

8. Page

- Represents the page of the application.
- Attributes:
  - `page_id`: Unique identifier for the landing page.
  - `title`: Title of the page.
  - `content`: Content displayed on the page.
  - `buttons`: Navigation buttons (e.g., "Login", "Sign Up").

User Facing App - Workflow Integration

1. Sign-Up Process :
    - The user fills out the `SignUpPage` form.
    - The ZIP code is validated via an API call.
    - If eligible, the user data is stored in the `User` class, and the user is redirected to the landing page or dashboard.
2. Profile Creation :
    - After signing up, the user's profile (`UserProfile`) is created automatically, displaying their name, profile picture, and survey progress.
3. Survey Interaction :
    - Users can view and complete surveys from their profile page.
    - Progress is tracked in the `UserProfile` class.
4. Authentication :
    - Users log in using credentials stored in the `User` class.
    - Session tokens are managed by the `Authentication` class.

NEW ZIP Code Validation Workflow

1. Step 1: User Input
    - The user enters their ZIP code in the designated field.
2. Step 2: Real-Time API Validation
    - As soon as the user finishes entering the ZIP code (or moves to the next field), an API call is triggered to validate the ZIP code.
    - The API checks the ZIP code against a database of eligible regions.
3. Step 3: Eligibility Check
    - If the ZIP code corresponds to an eligible region :
        - Should just assign eligibility status as "eligible" to the user record
        - Allow the user to proceed with the rest of the sign-up process.
    - If the ZIP code corresponds to a non-eligible region :
        - I want a user to still be able to complete sign up, they just should have their eligibility record set as non-eligible
        - Allow the user to proceed with the rest of the sign-up process.
4. Step 4: Assign Eligibility Status
    - If eligible, assign the user an "eligible" status in the backend database.

- If not eligible, assign the user an “non-eligible” status in the backend database

Answers to Clarifying Questions

1. What is the difference between a `user` and a `profile`? Why would there be two IDs for the same user?

- User : Represents the core identity of a person in the system. It contains essential information like `user_id`, `email`, `password`, **and other authentication-related details.**
- **Profile :** Represents additional information about the user that is not directly tied to authentication. This includes their progress in surveys, preferences, and **personal details like `first_name`, `last_name`, and `profile_pic_url`.**
- Why Two IDs?
  - The `user_id` is the primary key used to uniquely identify a user across the entire system (e.g., for authentication and linking to other tables).
  - The `profile_id` is specific to the `user_profile` table and is used to track profile-specific data. This separation allows for scalability and modularity:
    - If you need to add more profile-related features in the future, you can do so without affecting the core `user` table.
    - It also allows multiple profiles to exist for a single user in advanced systems (e.g., different profiles for different roles).

What does "progress: Map" mean?

- Progress: Map refers to a data structure that tracks the user's progress in completing surveys. In this context:
  - The `Map` could be implemented as a dictionary or key-value pair structure.
  -

```
{
  "survey_1": "completed",
  "survey_2": "in_progress",
```

```
  "survey_3": "not_started"
}
```

- Each key represents a `survey_id`, and the value indicates the user's progress status for that survey (e.g., "completed", "in_progress", "not_started").

Surveys - I had: `survey_id`, `survey_name`, and `survey_description`. What is `category`?

- Category :
  - The `category` attribute groups surveys into logical categories based on their purpose or theme. For example:
    - `"Basics"`: Surveys about general user information.
    - `"Relationship"`: Surveys about relationships or social preferences.
    - `"Political Views"`: Surveys about political opinions.
  - This helps organize surveys and makes it easier for users to find relevant ones.

What is `questions: List`?

- Questions: List :
  - This attribute represents a collection of questions associated with a specific survey.
  - Each item in the list corresponds to a `question` object (or record) that contains details like `question_id`, `question_text`, and `question_type`.

```
[
 {
  "question_id": "q1",
  "question_text": "What is your favorite color?",
  "question_type": "multiple_choice"
 },
 {
  "question_id": "q2",
  "question_text": "Rate your experience from 1 to 5.",
  "question_type": "rating_scale"
 }
]
```

All of the questions should have a `question_id` and map to the respective `survey_id`. In the Miro, it shows each question also has `question_text` and `question_type`.

- Yes, this is consistent with the UML diagram:
  - Each `question` has:

    - `question_id`: A unique identifier for the question.
    - `question_text`: The actual text of the question (e.g., "What is your favorite color?").
    - `question_type`: The type of question (e.g., `text_input`, `multiple_choice`, `rating_scale`).
- The `questions: List` in the `survey` class links each question to its respective `survey_id`. This relationship ensures that questions are grouped under the correct survey.

Sample Query Example for Responses

To query the database for responses matching specific criteria, you can use the following SQL query:

```
SELECT r.response_id
FROM response r
JOIN question q ON r.question_id = q.question_id
JOIN survey s ON q.survey_id = s.survey_id
WHERE s.survey_id = 'X' -- Replace 'X' with the desired survey ID
  AND q.question_id = 'Y' -- Replace 'Y' with the desired question ID
  AND r.response_value = 'Z'; -- Replace 'Z' with the desired response value
```

Key Relationships

1. User manages Authentication :
    - A user interacts with the `authentication` class for login, registration, and password recovery.
2. User owns UserProfile :
    - Each user has a profile that tracks their survey progress and displays their personal information.
3. UserProfile displays Surveys :
    - The profile page shows available surveys and their completion status.
4. Survey contains Questions :
    - Each survey is composed of multiple questions.
5. Question collects Responses :
    - Each question can have multiple responses from different users.
6. Response links User and Question :
    - A response is tied to a specific user (`user_id`) and a specific question (`question_id`).

UML Diagram for the Event Management

This UML diagram provides a clear structure for implementing the event management system based on the events requirements shared earlier.

Addressing Your Questions

1. Do we need an `audience_name` in the `audience_table`? This is `table_name`

Yes, adding an `audience_name` is a good idea for usability. It allows admins to easily identify and select audiences when creating events. Instead of referencing the audience by its `table_id`, they can use a human-readable name like `"Young Professionals"` or `"VIP Customers"`. This makes the system more intuitive.

2. How will we know if a user is in an audience table?

Explicitly store the `user_ids` of users who belong to each audience. This ensures that we can quickly determine which users are part of a specific audience.

- Why `user_ids`?
    - The `user_ids` field is a list of user IDs that match the `query_criteria`.

```
{
  "table_id": "t1",
  "table_name": "Young Professionals",
  "query_criteria": "age BETWEEN 25 AND 35 AND income > 50000",
  "user_ids": ["u1", "u2", "u3"]
}
```

3. Do we still need `query_criteria` in the `audience_table`?

Yes, `query_criteria` is still necessary because it defines how the audience was created. However, once the audience is created, the system should execute the query and populate the `user_ids` field. This ensures that:

- The `query_criteria` serves as a record of how the audience was filtered.
- The `user_ids` field provides a concrete list of users who belong to the audience.

4. How does this work when creating an event?

When creating an event, the admin selects an audience by its `table_name` (e.g., `"Young Professionals"`) instead of its `table_id`. Behind the scenes:

- The system retrieves the `user_ids` associated with the selected audience.
- Only users in the `user_ids` list will see the event.

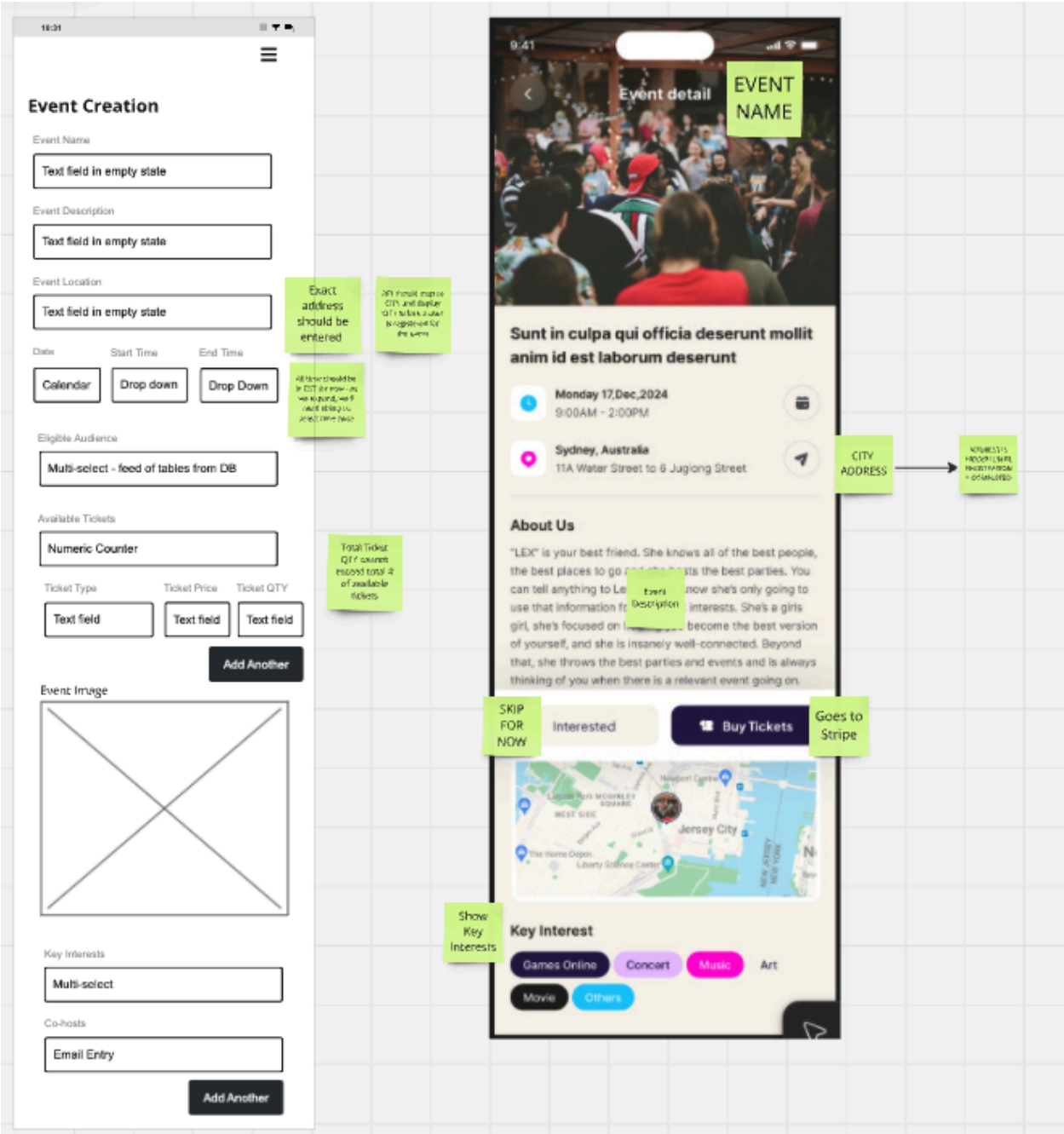
Explanation of Classes and Relationships - Event Management

1. User

- Represents a registered user in the system.
- Attributes:
  - `user_id`: Unique identifier for the user.
  - `first_name`, `last_name`: Personal details.
  - `email`: Email address used for login.
  - `password`: Encrypted password for authentication.
  - `mobile_number`: Phone number for communication.
  - `zip_code`: ZIP code for regional eligibility checks.
  - `profile_pic_url`: URL of the uploaded profile picture.
  - `eligibility_status`: Indicates whether the user is eligible to participate ("eligible" or "non_eligible").
- Relationships:
  - Linked to `registration` to track which events the user has registered for.

2. Event

- Represents an event with details like title, date, time, capacity, and ticket price.
- Attributes:
  - `event_id`: Unique identifier for the event.
  - `title`: Title of the event (e.g., "Summer Concert").
  - `date`: Date of the event.
  - `time`: Time of the event.
  - `capacity`: Total number of tickets available.
  - `description`: Description of the event.
  - `theme`: Theme or purpose of the event (optional).
  - `ticket_price`: Base ticket price (if applicable).
  - `created_by`: Admin ID who created the event.
  - `audience_table_id`: Links to the audience table for eligibility.
- Relationships:
  - Linked to `ticket_type` to define ticket types for the event.
  - Linked to `audience_table` to determine eligible users.
  - Linked to `registration` to track attendee registrations.



3. Ticket Type

- Represents different ticket types (e.g., General Admission, VIP) with specific pricing, quotas, and access levels.
- Attributes:
  - `ticket_type_id`: Unique identifier for the ticket type.
  - `event_id`: Links to the parent event.

    - `name`: Name of the ticket type (e.g., "VIP", "Early Bird").
    - `price`: Price of the ticket.
    - `quota`: Number of tickets available for this type.
    - `access_level`: Access level granted by this ticket type.
    - `available_tickets`: Remaining tickets for this type.
- Relationships:
    - Linked to `event` to associate ticket types with specific events.
    - Linked to `registration` to track which ticket type was purchased.

4. Registration

- Tracks attendee registrations, including the ticket type they purchased and their payment status.
- Attributes:
    - `registration_id`: Unique identifier for the registration.
    - `event_id`: Links to the event being registered for.
    - `ticket_type_id`: Links to the ticket type selected.
    - `user_id`: Links to the user who registered.
    - `payment_status`: Payment status (e.g., "pending", "completed").
    - `dietary_prefs`: Dietary preferences of the attendee.
    - `check_in_status`: Whether the attendee has been checked in.
    - `is_invited` :
        - Indicates whether the user was invited to the event.
        - Set to `True` when the user is part of the audience table linked to the event.
    - `is_registered` :
        - Indicates whether the user has completed the registration process (e.g., after payment confirmation).
        - Set to `True` when the user successfully registers for the event.
    - `has_attended` :
        - Indicates whether the user attended the event.
        - Set to `True` when the user checks in at the event (e.g., via QR code or manual check-in).
        -
- Relationships:
    - Links `user` to `event` via `registration`.

5. Audience Table

- Defines the audiences eligible to see specific events. It includes query criteria and a list of user IDs that match the criteria.
- Attributes:
  - `table_id`: Unique identifier for the audience table.
  - `table_name`: Human-readable name of the audience (e.g., "Young Professionals").
  - `query_criteria`: Criteria used to filter users into this table.
  - `user_ids`: List of user IDs who match the query criteria.
- Relationships:
  - Linked to `event` to determine which users are eligible for specific events.

6. Discount Code - This is populated via a Manual process, we just need the table created.

- Represents promotional codes for discounts on ticket purchases.
- Attributes:
  - `code_id`: Unique identifier for the discount code.
  - `code`: The actual discount code (e.g., "SAVE10").
  - `discount_type`: Type of discount (e.g., percentage off, fixed amount off).
  - `value`: Value of the discount.
  - `expiry_date`: Expiry date of the discount code.
  - `usage_limit`: Maximum number of times the code can be used.
- Relationships:
  - Can be linked to `event` or `ticket_type` to apply discounts to specific tickets.

Stripe Integration for Discount Codes

While Stripe handles payment processing, the database still needs to store discount codes for tracking purposes:

1. Discount Code Creation :
   - Admins create discount codes in the database with attributes like `code`, `discount_type`, `value`, `expiry_date`, and `usage_limit`.
2. Stripe Integration :
   - During checkout, the system validates the discount code against Stripe's API.
   - If valid, the discount is applied, and the usage count is updated in the database.
3. Tracking Usage :

    - The database tracks how many times each code has been used and enforces usage limits.

7. Feedback

- Captures post-event feedback from attendees.
- Attributes:
    - `feedback_id`: Unique identifier for the feedback.
    - `event_id`: Links the feedback to the specific event.
    - `rating`: Rating given by the attendee (e.g., 1–5 stars).
    - `comments`: Comments provided by the attendee.
- Relationships:
    - Linked to `event` to associate feedback with specific events.

8. Attendee

- Represents a user who registered and attended the event.
    - IAttributes:
    - `attendee_id`: Unique identifier for the attendee.
    - `user_id`: Links the attendee to their user account.
    - `name`: Name of the attendee.
    - `email`: Email address of the attendee.
    - `survey_responses`: Responses to surveys used for filtering into audience tables.
    - `personal_info`: Personal information about the attendee (e.g., age, income, preferences).
- Relationships:
    - Linked to `user` to associate attendees with registered users.
    - Linked to `registration` to track which events the attendee has registered for.

Key Table Relationships

1. Audience Table → Registration :
    - When an admin assigns an audience table to an event, the system creates a `registration` record for each user in the audience (`user_ids`) and sets `is_invited = True`.
2. User → Registration :

    - A user registers for an event by completing the registration process. The system updates the `is_registered` field to `True`.
3. Registration → Event :
    - The `registration` table links users to specific events and tracks their statuses (`is_invited`, `is_registered`, `has_attended`).
4. Feedback → Registration :
    - Feedback is linked to the `event_id` and indirectly to the `registration` record via the `user_id`.

Example Workflow

Step 1: Invite Users

- When an admin creates an event and selects an audience table:
    - The system retrieves the `user_ids` from the `audience_table`.
        - is_invited = True
        - is_registered = False
        - has_attended = False

Step 2: User Registers

- When a user registers for an event:
    - The system updates the `is_registered` field in the `registration` table to `True`.

Step 3: User Attends

- When a user checks in at the event:
    - The system updates the `has_attended` field in the `registration` table to `True`.

Step 4: Restrict Access to Event Location

- To ensure only registered users can see the full event location:
    - Query the `registration` table to check if `is_registered = True` for the user and event.

UML Key Relationships - Event Management

1. Admin manages Events :
    - An admin creates and configures events, including selecting audiences and ticket types.
2. Event uses AudienceTable :
    - Each event is associated with an audience table to define visibility.
    - Only users in the `user_ids` list of the selected audience can see the event.
3. Event offers TicketTypes :
    - Events can have multiple ticket types with different prices, quotas, and access levels.
4. Attendees register for Events :
    - Attendees are linked to events via registrations.
    - Registrations track ticket type, dietary preferences, and check-in status.
5. AudienceTable filters Attendees :
    - The `query_criteria` determines which users belong to an audience.
    - The `user_ids` field ensures accurate filtering of eligible users.
6. DiscountCode applies to Tickets :
    - Discount codes are applied during ticket purchase and tracked in the database.

Ticket Purchases - Stripe Checkout Integration

Instead of manually creating a payment page for each event, we can dynamically generate a Stripe Checkout Session for each event at runtime. This eliminates the need to pre-create or share static links during the event creation process - [How Checkout works | Stripe Documentation]

1. Dynamic Stripe Checkout Sessions

Stripe provides an API called **Checkout Sessions** that allows us to create a payment page on-the-fly when a user clicks "Buy Tickets." This approach is highly dynamic and integrates seamlessly with the backend.

Steps to Implement Dynamic Checkout:

1. Event Creation in Admin Panel :
    - When an admin creates an event, they configure ticket types (e.g., General Admission, VIP) and prices.

    - The system stores these details in the database but does not create a Stripe link yet.
2. User Clicks "Buy Tickets" :
    - When a user clicks the "Buy Tickets" button for a specific event, your backend generates a Stripe Checkout Session dynamically.
    - The session includes:
        - Event-specific details (e.g., event name, ticket type, price).
        - Success and cancel URLs (where users are redirected after payment or cancellation).
3. Redirect to Stripe Checkout :
    - The backend returns the `session.id` from Stripe to the frontend.
    - The frontend redirects the user to the Stripe-hosted checkout page using the `session.id`.
4. Payment Confirmation :
    - After the payment is completed, Stripe redirects the user to the success URL.
    - Your backend listens for a webhook from Stripe to confirm the payment and update the database (e.g., marking tickets as purchased).

How It Works for Multiple Events

The key idea is that each event has its own unique `eventId`, and the backend dynamically generates a Stripe Checkout Session specific to the event when a user clicks "Buy Tickets." This ensures that the payment process is tailored to the selected event and ticket type, without requiring pre-created links or static configurations.

Steps for Handling Multiple Events

1. Event Creation

- When an admin creates an event in the system:
    - They define details like `event_id`, `title`, `date`, `ticket types`, and `prices`.
    - These details are stored in the database (e.g., in an `events` table).

2. Dynamic Checkout Session

- When a user clicks "Buy Tickets" for a specific event:
    - The frontend sends the `eventId` and `ticketTypeId` to the backend.

    - The backend fetches the event and ticket details from the database and uses them to create a Stripe Checkout Session dynamically.

3. Stripe Integration

- Each Stripe Checkout Session is unique to the event and ticket type. For example:
    - Event A might have tickets priced at $50 for General Admission and $100 for VIP.
    - Event B might have tickets priced at $75 for General Admission and $150 for VIP.
- The backend dynamically sets these prices and other details (e.g., event name) in the Stripe API request.

4. Scalability

- Since the logic is dynamic, you can handle as many events as needed without modifying the core implementation.
- Each event operates independently, and the system scales seamlessly.

Advantages of Dynamic Checkout:

- No Pre-Created Links : You don’t need to manually create or manage payment links for each event.
- Scalability : Works seamlessly for any number of events and ticket types.
- Customization : Each checkout session can include event-specific details like ticket type, price, and event name.
- Security : Payment processing happens entirely on Stripe’s secure servers, reducing your PCI compliance burden.