052400-1 VU Information Management and Systems Engineering (2025S)

Milestone 2

(Please note: submission deadline: Tue 16.06.2025 13:00)

Group X

Student 1: Jaupi, Joana 12448983

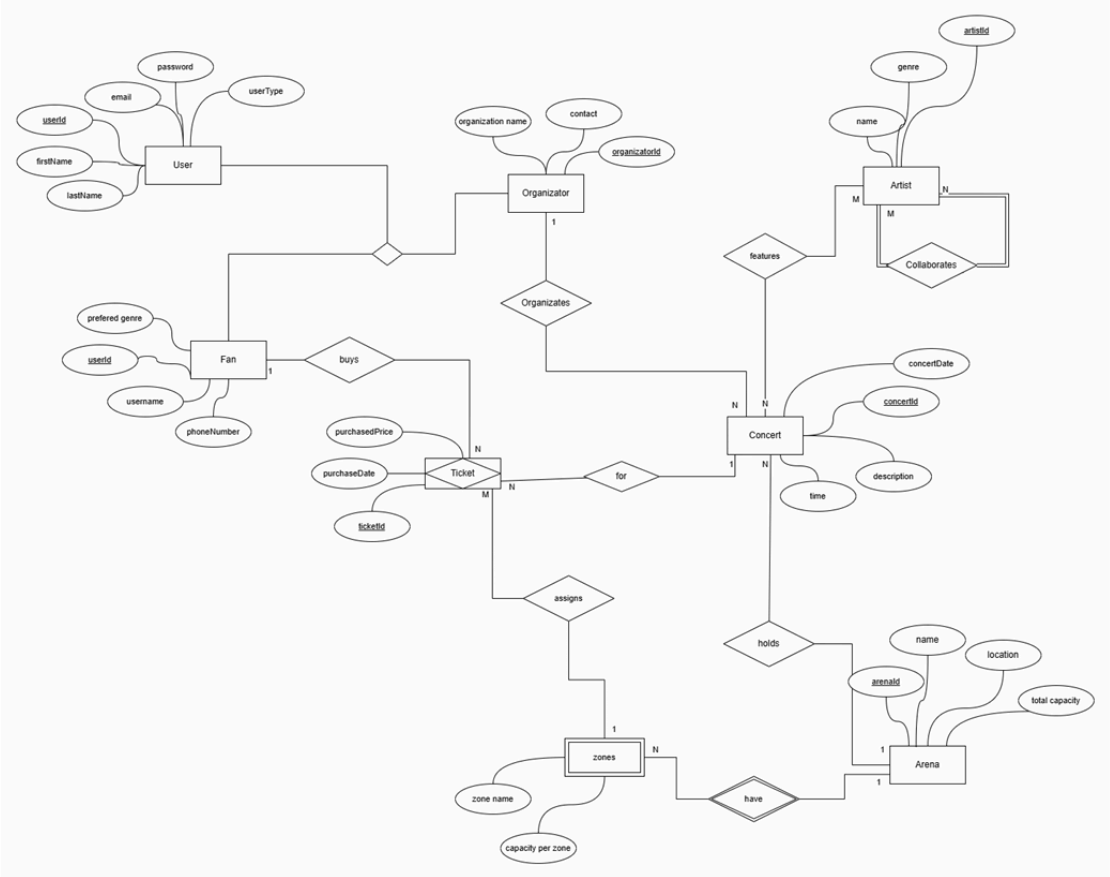
Student 2: Talelli, Livia, 12449604

16.06.2025, Vienna

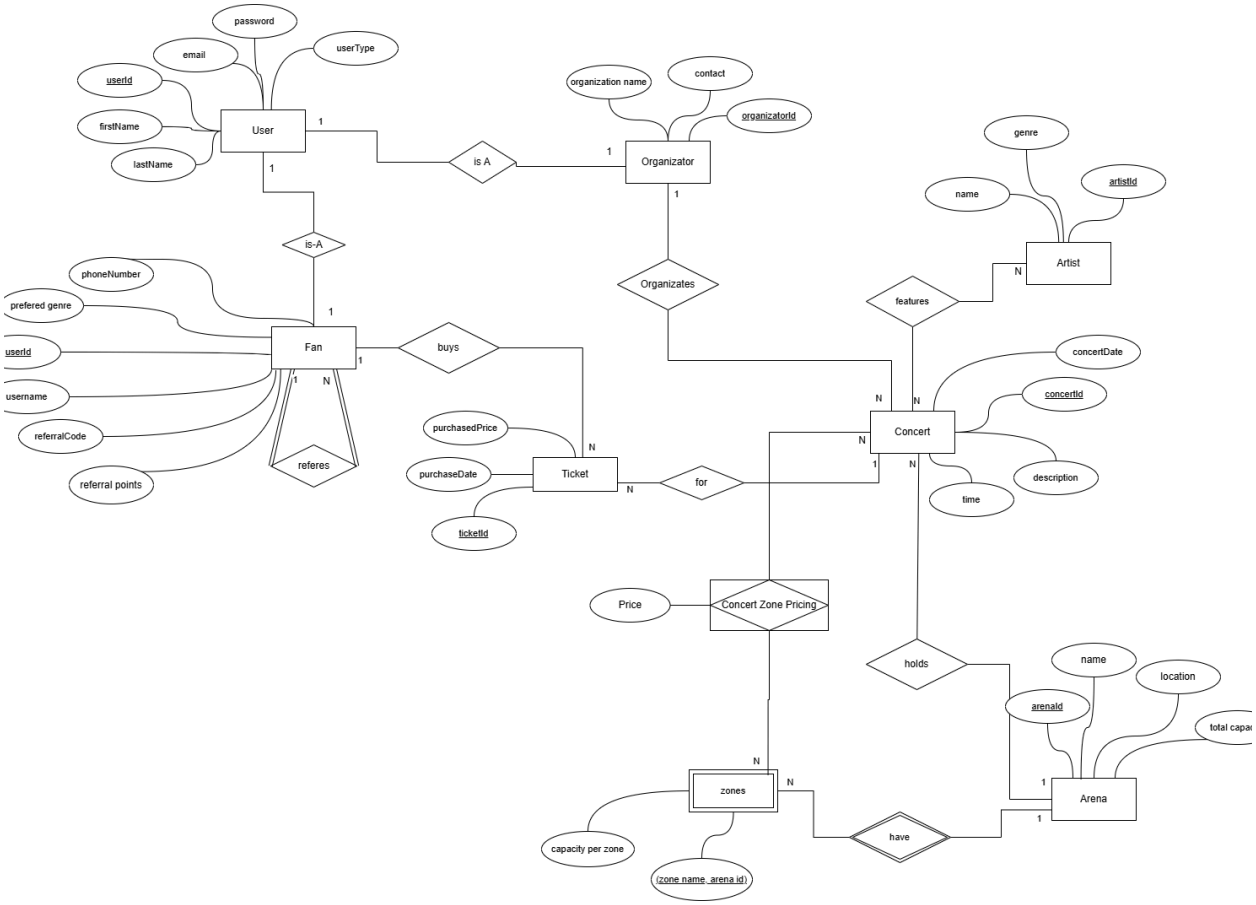
1 Milestone 1 – Recap and Revisions

1.1 Recap: ER-Diagram

ER-Diagram v1 (from Milestone 1):



Revised ER Diagram:



1.2 Revisions and Changes

Application Domain:

- The artist collaboration unary relationship was removed from the data model as it falls outside the current application scope and does not align with the intended system behavior. Conceptually, modeling artist collaboration as a **unary relationship** (i.e., artist collaborates with another artist) would inaccurately reflect real-world scenarios. In practice, artist collaborations are typically **concert-specific** and not global or persistent across all time. A unary relationship in this case would introduce an artificial abstraction, misrepresenting how collaborations actually occur.

  While we considered modeling collaboration as a relationship specific to a concert (e.g., through a ternary or associative entity), doing so would violate the project guidelines—specifically, the requirement to include **at least one unary relationship** and a maximum of **nine entities**. To maintain conceptual integrity and meet the constraints, we chose to omit the collaboration relationship entirely.

Instead, we introduced a **"Fan references other fans"** unary relationship. This models a referral system where one fan can refer to multiple others using a **referral code**, granting the referred fan a **10% discount**. For each successful referral, the referring fan earns **points (5 points per ticket bought from their referral)** that can be redeemed for discounts on future bookings. **(up to 50 points per purchase)**

- Introduced an associative entity **concert_zone_pricing** as prices for tickets in concerts were missing in the last model, and ticket prices are **zone** based
- Specified the correct cardinalities for the relationships:

  1) Between **Artist** and **Concert**, indicating how many artists can perform at a concert and how many concerts an artist can participate in;
  2) Between **Ticket** and **Fan**, indicating that each ticket is owned by one fan, while a fan can purchase multiple tickets;
  3) Between **Zone** and **Arena**, indicating that each zone belongs to a single arena, but an arena can contain multiple zones.

Logical Design:

- Updated the ER diagram to follow proper Chen notation standards.
- Fixed modeling of associative entities (used for many-to-many relationships) and corrected the **Zone** weak entity by defining in the ER diagram a composite primary key — consisting of its own identifier and the primary key of the **Arena** it belongs to

Relational Modeling:

- Added the missing primary keys in the textual description to the **Artist**, **Concert**, and **Ticket** tables in the relational schema: **artist_id** for **Artist**, **concert_id** for **Concert**, and **ticket_id** for **Ticket**.

Individual changes: (Jaupi, Joana, 12448983)

In the Use Case Definition and Design:

Feedback by professor and my changes -

1.2.1 Student 1 Textual Description

- Step missing: Loading concerts

Added in the textual description of the use case. (see below)

- Step missing: Selecting zone

Added in the textual description of the use case (see below)

"The fan has a valid payment method or sufficient balance." This precondition is not supported by your model

From the use case I have removed the precondition of “a fan has a valid payment method or sufficient balance”.

- If tickets are created and inserted into the DB after purchase, and the price is saved in the ticket, you have no opportunity to display the price beforehand.

That is why we have introduced a ‘concert_zone_pricing’ where the ticket prices are available before purchasing a ticket.

- "Electronic ticket is generated": If you are not going to realize this, remove it

I did implement it. Fans can download the electronic ticket (a PDF of the ticket)

- "Backend sends an email with a ticket summary.” If you are not going to realize this, remove it

Revised Use case with the changes of the model

Preconditions:

- The user is authenticated as a fan.
- The concert has available tickets in the selected Zone.

Main Flow:

- Fan logs into the system.
- Fan browses upcoming concerts and selects a concert to attend.
- System loads and displays a list of upcoming concerts.
- Backend calculates remaining zone capacities for the selected concert.
- System displays concert details including (arena, date and time, available zones within the arena, capacity per zone, ticket prices per zone)
- Fan selects a preferred available zone (eg. “Zone VIP”) for the selected concert.
- Fan specifies the number of tickets.
- Backend validates:
  - The requested number of tickets is valid. (not more than the remaining zone capacity)
- (optional) Fan applies a referral code
  - Backend validates if the fan has already used a referral code.
  - Backend validates the validity of the referral code.
  - If the referral code is validated, a discount of 10% is applicable.
- If a purchase is confirmed by the fan, the backend inserts as many tickets as the quantity was specified by the fan, and the price with or without discount, into the tickets table.
- The system sends confirmation to the frontend.
- Frontend displays a success screen and redirects fans to his tickets page.
- Backend sends an email with a ticket summary.

Postconditions*:*

- A new ticket is inserted into the tickets table.
- Customer receives ticket booking confirmation.
- Electronic ticket is generated
- Capacity in selected zone is updated

Individual changes:

STUDENT 2 (Talelli, Livia)

1- Use Case Definition and Design:

Based on the Milestone 1 feedback , the statement **"The organizer has the necessary rights to organize and manage concerts."** is misleading because the model does not define different permissions for the organizer role. To meet the requirements , I removed the statement from the Preconditions.

2- Textual Description:

To address the feedback for a more precise trigger and clearer creation steps, the use case has been revised with the following improvements:

Changes Made:

- **Trigger clarified**:
  The trigger is now explicitly defined as: The organizer clicks the **"Create New Concert"** button to start the concert creation process.
- **Detailed creation flow added**:
  The organizer provides the **concert description**, **date**, and **start time**.
- **Backend filtering logic explained**:
  Based on the selected date and time, the backend filters and retrieves only the **arenas** and **artists** that are available on the selected concert date from the database. These filtered arenas and artists are then shown to the organizer.
- **Step-by-step selection process described**:
  - The organizer selects an **available arena**.
  - The organizer **configures zone pricing**.
  - The organizer **selects artists** from a list of available ones (only selection is allowed, not adding new artists).
- **Removed collaboration feature**: The step where the organizer could specify whether artists perform **solo or in collaboration** has been **removed**, as collaboration is no longer a feature in the system.
- **Concert creation**:

- The data is **inserted into the database**.

- Finally, the organizer is presented with the **newly created concert and its full details**.

Trigger: The organizer clicks the **"Create New Concert"** button to start the concert creation process.

Main Flow:

1. The organizer clicks the **"Create New Concert"** button.
2. The organizer enters a **concert description**.
3. The organizer selects a **date** for the concert.
4. The organizer enters the **start time** for the concert.
5. The **backend** receives the selected date , and **requests all available arenas** from the **database**.
6. The **database** responds with arenas that are **free at the selected date**.
7. The **backend filters** and displays **only available arenas** to the organizer.
8. The organizer selects one of the available arenas.
9. The organizer proceeds to **configure zone pricing** for the selected arena.
10. The **backend** requests a list of **available artists** from the **database**.
11. The **database** returns all artists that are available for the selected date.
12. The **backend** displays these artists to the organizer.
13. The organizer selects one or more **artists** to perform at the concert from the available list.
14. The organizer confirms the creation of the concert.
15. The new concert information is inserted into the **database**.
16. The **backend fetches** all concerts created by the organizer from the database.
17. The newly created concert and its details are shown to the organizer.

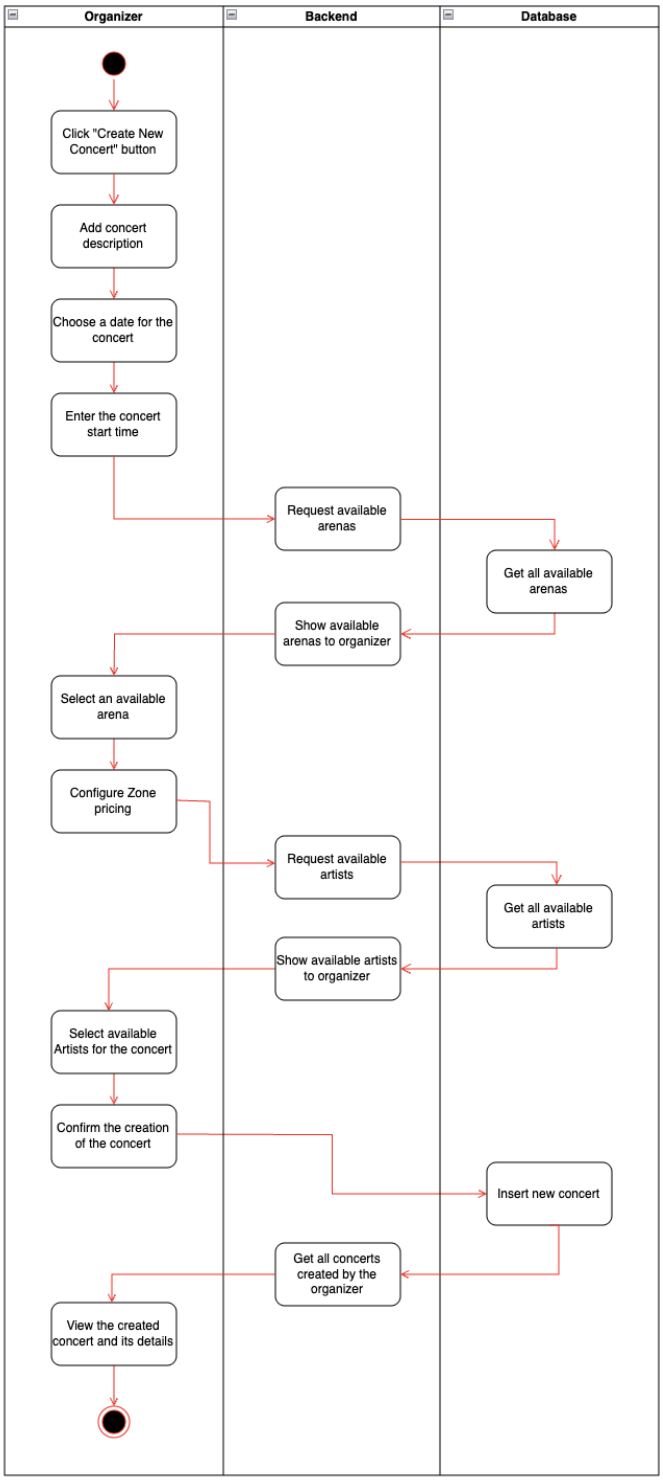
Postconditions:

1. The new concert is successfully registered in the system database.

2. The concert details (description, date, time, arena, zones, and artists) are stored and visible to the organizer.

3. The concert becomes accessible for customer booking.

Graphical Representation:



2 Milestone 2

2.1 Infrastructure

Our application is built on a containerized, multi-service architecture orchestrated with Docker Compose, ensuring a consistent and reproducible environment for both **development** and **production**. The infrastructure is comprised of four main services that communicate over a dedicated bridge network for enhanced security:

- **Frontend Service**: A modern React single-page application that provides the user interface. It is responsible for all client-side rendering and interaction. In the production environment, it is served over HTTPS on port 443, utilizing self-signed SSL certificates for secure data transmission.
- **Backend Service:** A robust Node.js application that serves as the core API for the frontend. It handles business logic, data processing, and communication with the databases.
- **PostgreSQL Database**: A relational database instance (Postgres 16) used for storing structured data. The database schema is automatically initialized on the first run, and its data is persisted across container restarts using a dedicated Docker volume.
- **MongoDB Database:** A NoSQL database (Mongo 7) employed for storing less structured data, providing flexibility for various data models within the application. Similar to PostgreSQL, its data is persisted using a Docker volume.
- **(Only in Development Environment) pgAdmin (`pgadmin`)** A web-based PostgreSQL administration tool accessible at localhost:5050, useful for visual inspection and manual queries on the PostgreSQL database, mainly used during development.

This containerized setup encapsulates all dependencies and compilation steps within the Docker build process, allowing the application to run on any machine with Docker installed with minimal setup.

To ensure a clean separation between local development and production deployment, our application uses two distinct Docker Compose configurations: `docker-compose.dev.yml` and `docker-compose.prod.yml`. This differentiation allows us to tailor each environment to its specific use case. The **development environment** includes additional tools such as pgAdmin for database inspection and runs services with settings optimized for debugging and rapid iteration (e.g., HTTP, verbose logging). In contrast, the **production environment** focuses on security and performance, serving the frontend over **HTTPS** with self-signed SSL certificates and excluding development-only services.

Running the Application

To launch the application, ensure you have Docker and Docker Compose installed on your system. Then, follow these simple steps:

Extract the contents of the submitted project .zip file. Open a terminal or command prompt and navigate to the root directory of the extracted project (the folder containing the docker-compose.prod.yml file). Execute the following command to build the container images and start all the services in the background:

(production environment) Run:

```
docker-compose -f docker-compose.prod.yml up -d --build
```

(development environment) Run:

```
docker-compose -f docker-compose.dev.yml up -d --build
```

Once the command finishes, the application will be accessible in your web browser at https://localhost (production) or http://localhost:3000 (dev)

Note: In production As the project uses self-signed SSL certificates for HTTPS, your browser will likely display a security warning. You can safely proceed past this warning to access the application.

2.2 RDBMS Implementation

2.2.1 DB Setup / Data Import / Base Function of Web System

The application's relational persistence layer is built on PostgreSQL. The database schema is designed to model the core entities of a concert booking platform, enforcing data integrity through relational constraints.

The process for establishing the database is a deliberate two-phase approach that separates the initial, one-time schema creation from the flexible and on-demand data population.

Schema Initialization: When the application is first launched via `docker-compose`, the PostgreSQL service automatically executes the `01-init.sql` script. This one-time setup creates the entire database schema, including all tables, relational constraints (primary and foreign keys), and server-side business logic like **triggers**. **This ensures that a correctly structured but empty database is always available the first time the environment is created.**

On-Demand Data Seeding via API Endpoint: The backend provides a dedicated API endpoint specifically for seeding the database.

Invoking this endpoint executes the logic contained within the `seed-data.ts` script. This script connects to the database and programmatically populates it with a rich set of mock data using the `Faker.js` library. The process includes:

1. **Clears Existing Data:** It first truncates all tables to ensure a clean slate, preventing data duplication on subsequent runs.
2. **Generates Users:** It creates 50 sample users with dynamically generated names and emails, securely hashing their passwords.
3. **Assigns Roles:** These users are then partitioned into 40 fans and 10 organizers, populating the respective tables with role-specific details like usernames and organization names.
4. **Populates Core Catalogs:** The script seeds the database with a variety of artists and arenas, complete with associated zones for each arena.
5. **Creates Events and Relationships:** Finally, it generates a set of concerts hosted by the sample organizers at various arenas. It then creates the necessary links for each concert, assigning artists to the lineup (`concert_features_artists`) and setting ticket prices for each zone (`concert_zone_pricing`). It even simulates a purchase history by creating sample tickets bought by fans.

The seeding script is only triggered through the endpoint `/api/admin/seed`

CHANGES TO RELATIONAL MODEL:

1. Referral System for Fans

- **New Additions:**
  - `referred_by` (self-referencing FK to `fans.user_id`)
  - `referral_code`, `referral_points`, `referral_code_used`
- These fields enable a referral tracking system, which was not present in the original schema, and introduce a unary relationship within the `fans` table.

2. Enhanced UUID Handling

- Old Schema: Used `UUID()` (MySQL-style) for default `CHAR(36)` primary keys.
- New Schema: Explicitly enables and uses PostgreSQL's `uuid-ossp` extension and `uuid_generate_v4()` function for uniform and reliable UUID generation.

3. Introduction of Zone-Based Dynamic Pricing

- **New Table:** `concert_zone_pricing(concert_id, arena_id, zone_name, price)`
- Replaces the hardcoded `purchase_price` logic from the `tickets` table, enabling zone- and concert-specific ticket pricing.

- The `tickets` table now references `concert_zone_pricing` via a composite foreign key, enforcing price consistency.

4. Tickets Table Refactor

- **Old Schema:** `purchase_price` stored directly in `tickets`, which could lead to redundancy.
- **New Schema:**
    - `purchase_price` retained but now validated against `concert_zone_pricing`.
    - More robust referential integrity via a new composite FK to `concert_zone_pricing`.

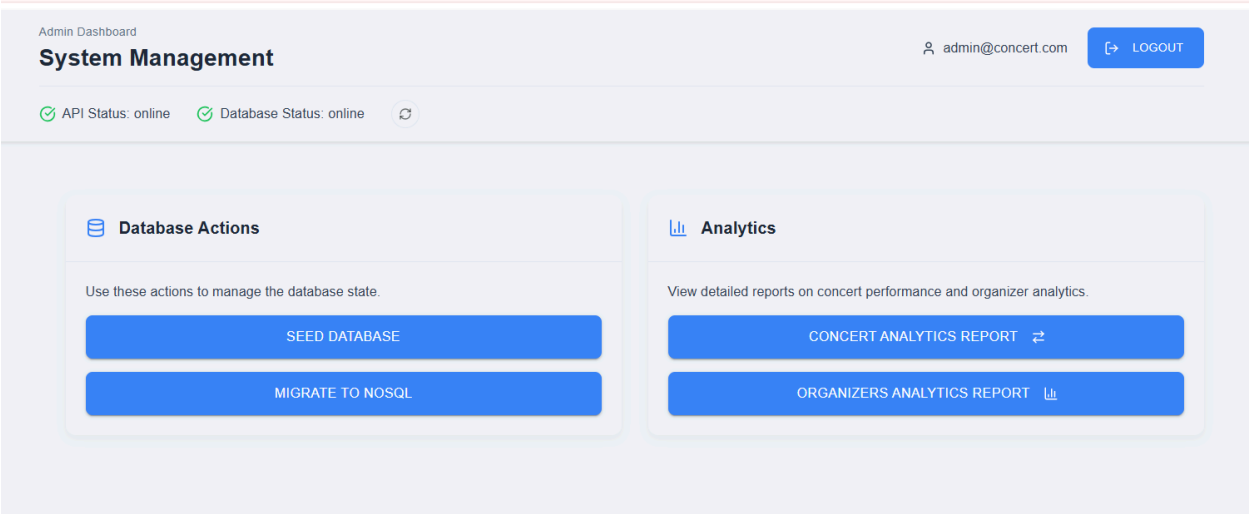
5. Removed Entity: `artist_collaborations`

- The table `artist_collaborations`, representing artist-artist relationships, has been dropped, due to not mirroring real life, since artist collaborations are per concert based, which if we had to implement, would go against the guidelines.

Base Function of Web System

The core functionality (without use cases) of our web application revolves around a **multi-role user system** that supports three distinct user types: **Admin, Organizer, and Fan**. The system employs **JWT-based authentication** to ensure secure login and logout processes, with **passwords securely hashed using bcrypt** to protect user credentials. It implements **role-based access control**, ensuring that each user only has access to features relevant to their role. The platform provides essential features such as **user registration and login**, forming the foundation for personalized and secure user experiences across different roles.

In addition to core authentication and role management, the base functionality of our web system includes a dedicated **admin panel** designed for superusers. Through this panel, administrators can **seed the PostgreSQL database with test data** and initiate **data migration from PostgreSQL to MongoDB**. The admin panel also provides access to **analytics reports**.



Student 1 Jaupi, Joana 12448983

2.2.2 Implementation of the Use Case and the Analytics Report

The application follows a layered architecture, separating concerns across `controllers, services, repositories, and DTOs`. Controllers handle requests and delegate business logic to `services`, which coordinate operations and enforce rules. Services interact with `repositories` that abstract database access, allowing the system to support both MongoDB and PostgreSQL through a `factory pattern` that selects the appropriate implementation. `Data Transfer Objects (DTOs)` are used to define the structure of data exchanged between layers, ensuring clarity and type safety. This design promotes modularity, flexibility, and ease of maintenance.

Use Case: Fan Ticket Purchase

Goal:
The goal of this use case is to allow **Fans** to seamlessly purchase tickets for their preferred concerts through the web system.

Steps Involved:

1. Fan signs up or logs in
2. Browses available concerts
3. Selects a specific concert
4. Chooses a seating zone
5. Optionally redeems loyalty points or referral codes *(new since Milestone 1)*
6. Completes the ticket purchase

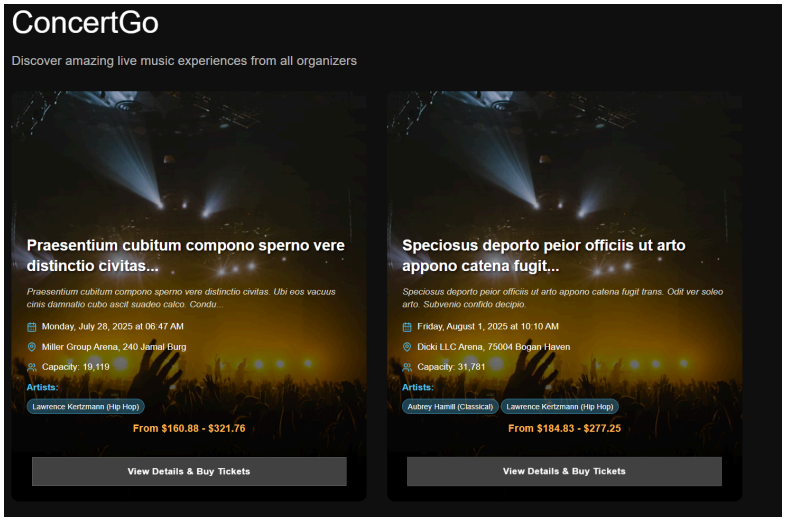
7. Receives email confirmation
8. Accesses an electronic version of the ticket (PDF) via their fan panel

Outcomes:

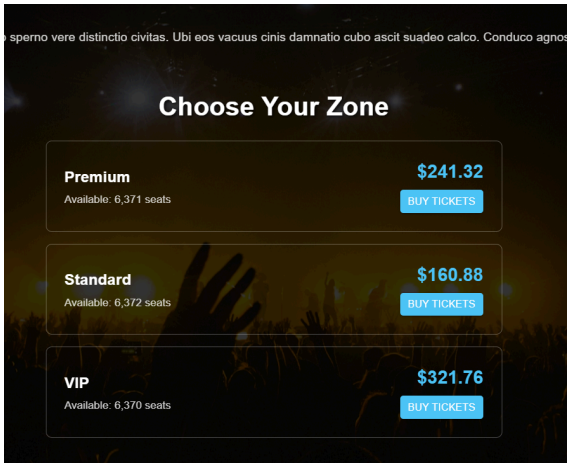
- A ticket is successfully created and linked to the fan's profile.
- Loyalty point balances or referral code usage are updated accordingly.
- An email confirmation is sent to the fan.
- The purchased ticket becomes available for download as a PDF.
- The ticket sale directly contributes to updated analytics and system records.

Screenshots from the steps of the usecase:

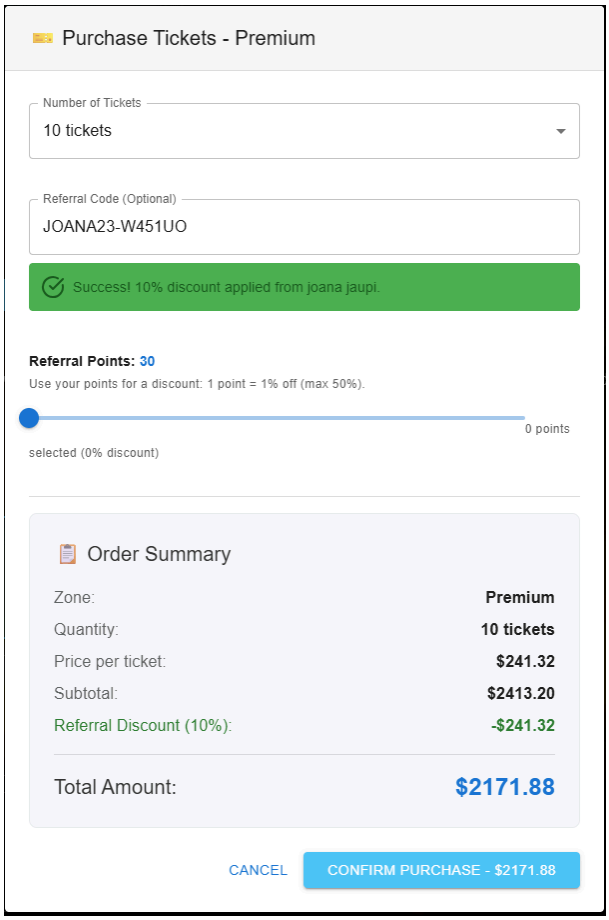
1. Concert Browsing

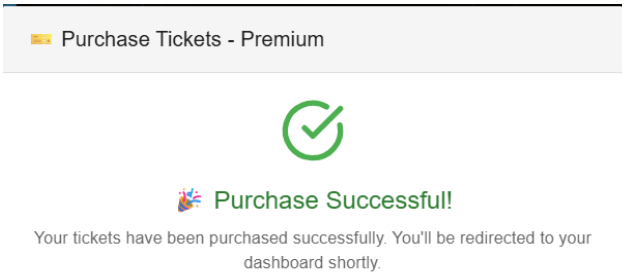


2. Concert Selection and Zone Selection

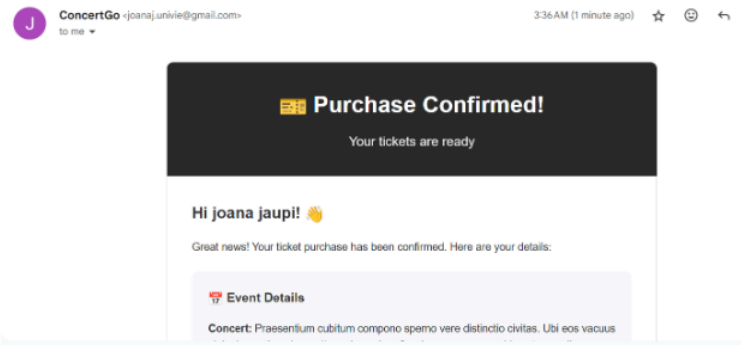


3. Tickets purchasing and applying referral code or referral points if applicable (a fan cannot use both)

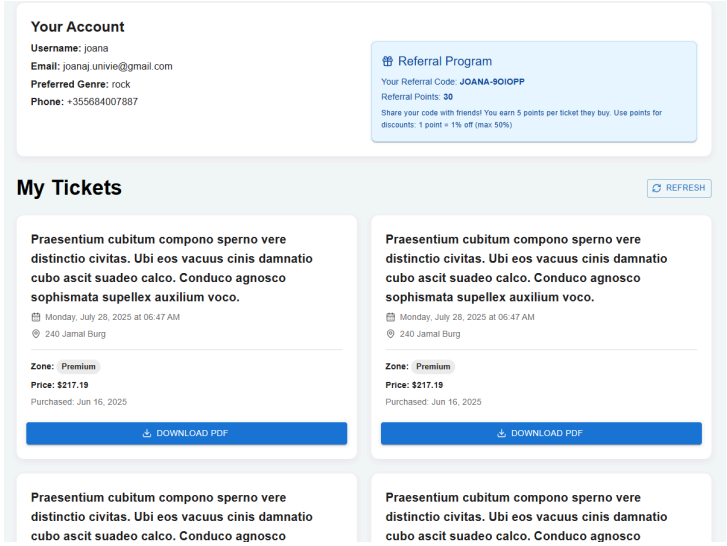




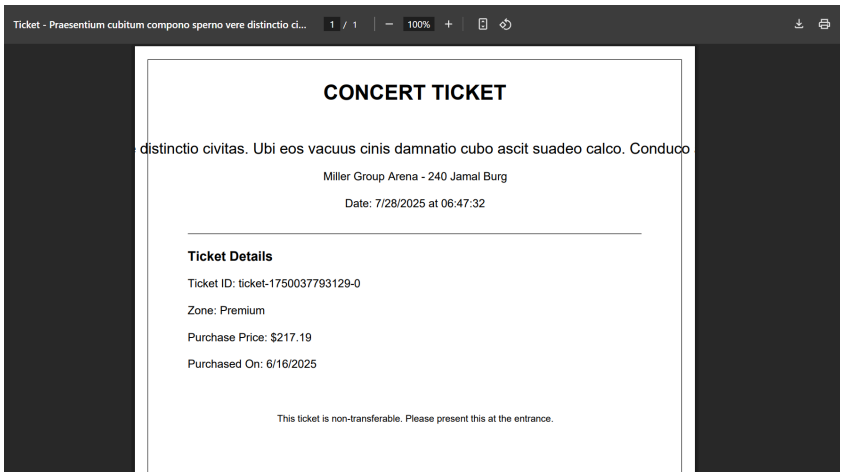
4. Confirmation of purchase through email



5. Fans can access their tickets through the fan panel.



6. Fans can generate electronic tickets (PDF)



Analytics Report: Ticket Sales for Upcoming Concerts

Goal:

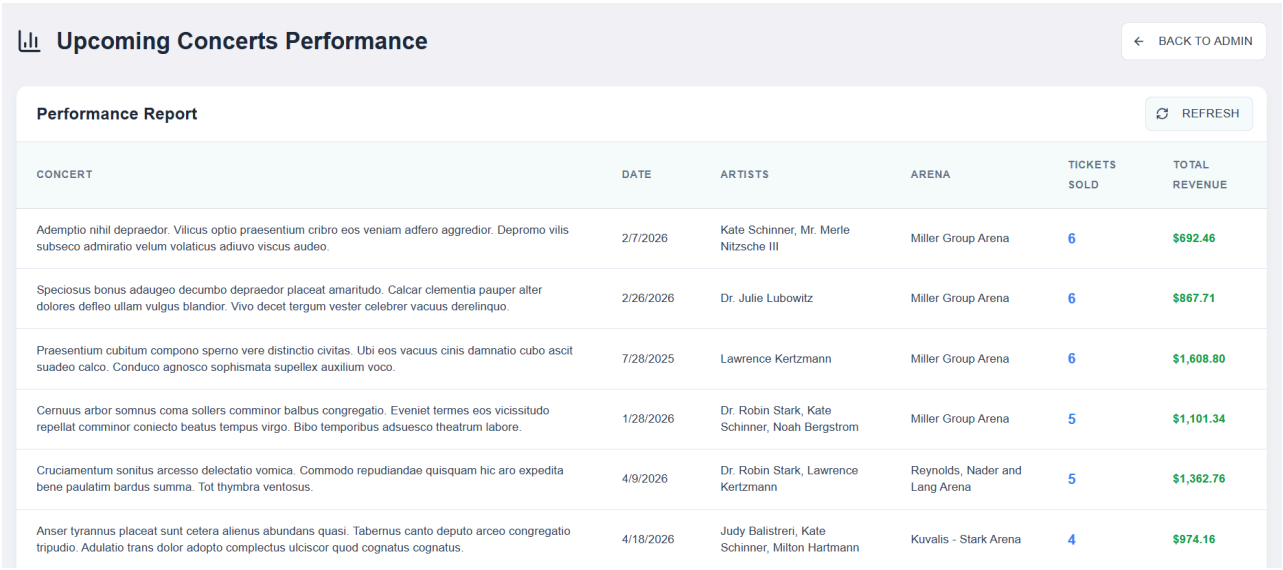
To provide performance insights into **upcoming concerts** by displaying the following metrics:

- Number of tickets sold
- Concert date, description, and venue
- Artists performing
- **Revenue generated** *(new addition since Milestone 1)*

Outcomes:

- When a fan purchases a ticket, the report automatically updates:
    - **Ticket count** for the selected concert increases
    - **Revenue** figure adjusts based on the ticket price and any discounts from loyalty/referral programs
- Enables the **admin** to identify which concerts are most successful, which venues attract the largest audiences, and which artists have strong fan engagement

Analytics Report Before Use Case has been performed:



Upcoming Concerts Performance

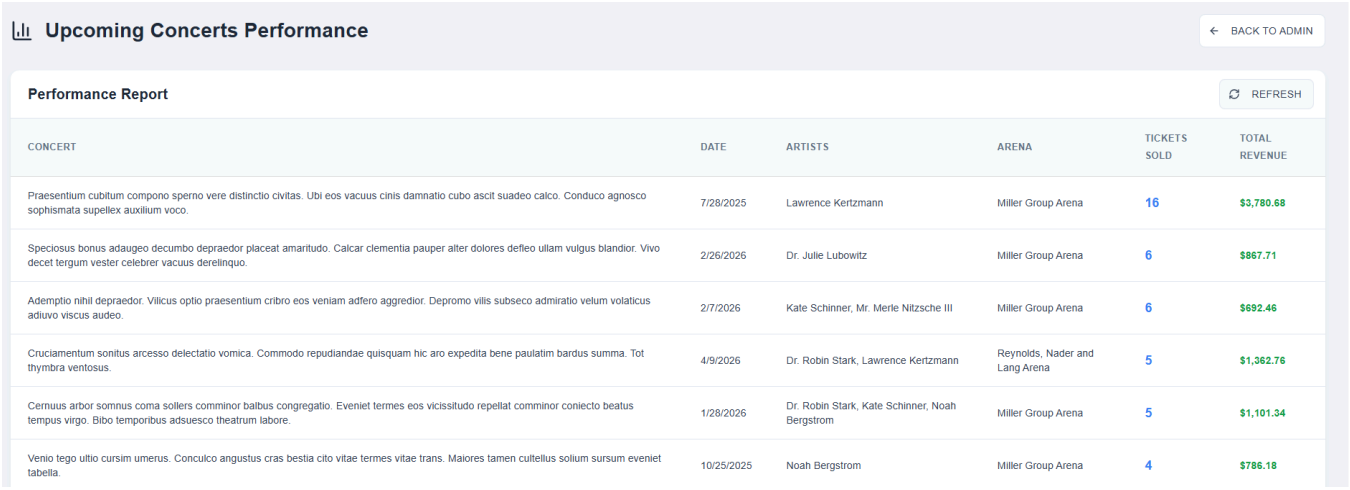
BACK TO ADMIN

Performance Report

REFRESH

CONCERT	DATE	ARTISTS	ARENA	TICKETS SOLD	TOTAL REVENUE
Ademptio nihil depraedor. Vilicus optio praesentium cribro eos veniam adfero aggredior. Depromo vilis subseco admiratio velum volaticus adiuvo viscus audeo.	2/7/2026	Kate Schinner, Mr. Merle Nitzsche III	Miller Group Arena	6	$692.46
Speciosus bonus adaugeo decumbo depraedor placeat amaritudo. Calcar clementia pauper alter dolores defleo ullam vulgus blandior. Vivo decet tergum vester celebrer vacuus derelinquo.	2/26/2026	Dr. Julie Lubowitz	Miller Group Arena	6	$867.71
Praesentium cubitum compono sperno vere distinctio civitas. Ubi eos vacuus cinis damnatio cubo ascit suadeo calco. Conduco agnosco sophismata supellex auxilium voco.	7/28/2025	Lawrence Kertzmann	Miller Group Arena	6	$1,608.80
Cernuus arbor somnus coma sollers comminor balbus congregatio. Eveniet termes eos vicissitudo repellat comminor coniecto beatus tempus virgo. Bibo temporibus adsuesco theatrum labore.	1/28/2026	Dr. Robin Stark, Kate Schinner, Noah Bergstrom	Miller Group Arena	5	$1,101.34
Cruciamentum sonitus arcesso delectatio vomica. Commodo repudiandae quisquam hic aro expedita bene paulatim bardus summa. Tot thymbra ventosus.	4/9/2026	Dr. Robin Stark, Lawrence Kertzmann	Reynolds, Nader and Lang Arena	5	$1,362.76
Anser tyrannus placeat sunt cetera alienus abundans quasi. Tabernus canto deputo arceo congregatio tripudio. Adulatio trans dolor adopto complectus ulciscor quod cognatus cognatus.	4/18/2026	Judy Balistreri, Kate Schinner, Milton Hartmann	Kuvalis - Stark Arena	4	$974.16

Analytics Report After Use Case has been performed (We bought 10 tickets):



Upcoming Concerts Performance

BACK TO ADMIN

Performance Report

REFRESH

CONCERT	DATE	ARTISTS	ARENA	TICKETS SOLD	TOTAL REVENUE
Praesentium cubitum compono sperno vere distinctio civitas. Ubi eos vacuus cinis damnatio cubo ascit suadeo calco. Conduco agnosco sophismata supellex auxilium voco.	7/28/2025	Lawrence Kertzmann	Miller Group Arena	16	$3,780.68
Speciosus bonus adaugeo decumbo depraedor placeat amaritudo. Calcar clementia pauper alter dolores defleo ullam vulgus blandior. Vivo decet tergum vester celebrer vacuus derelinquo.	2/26/2026	Dr. Julie Lubowitz	Miller Group Arena	6	$867.71
Ademptio nihil depraedor. Vilicus optio praesentium cribro eos veniam adfero aggredior. Depromo vilis subseco admiratio velum volaticus adiuvo viscus audeo.	2/7/2026	Kate Schinner, Mr. Merle Nitzsche III	Miller Group Arena	6	$692.46
Cruciamentum sonitus arcesso delectatio vomica. Commodo repudiandae quisquam hic aro expedita bene paulatim bardus summa. Tot thymbra ventosus.	4/9/2026	Dr. Robin Stark, Lawrence Kertzmann	Reynolds, Nader and Lang Arena	5	$1,362.76
Cernuus arbor somnus coma sollers comminor balbus congregatio. Eveniet termes eos vicissitudo repellat comminor coniecto beatus tempus virgo. Bibo temporibus adsuesco theatrum labore.	1/28/2026	Dr. Robin Stark, Kate Schinner, Noah Bergstrom	Miller Group Arena	5	$1,101.34
Venio tego ultio cursim umerus. Conculco angustus cras bestia cito vitae termes vitae trans. Maiores tamen cultellus solium sursum eveniet tabella.	10/25/2025	Noah Bergstrom	Miller Group Arena	4	$786.18

Student 2 Talelli, Livia, 12449604

2.2.2 Implementation of the Use Case and the Analytics Report

Use Case: Organizer Creates a Concert

Goal: The goal of this use case is to enable organizers to schedule and configure new concert events by entering essential information such as the concert description, date, start time, selected arena, zones pricing and participating artists.

Steps Involved:

- Organizer logs into the system
- Organizer clicks the **"Create New Concert"** button
- Enters concert details (description, date, and start time)
- Backend filters and displays only arenas available at the selected time
- Organizer selects an available arena
- The system loads zones for the selected arena
- Organizer enters pricing for every available zone
- Backend filters and displays only available artists
- Organizer selects one or more participating artists
- Organizer reviews all entered data
- Organizer confirms and submits the concert setup
- System saves the concert information
- The organizer is shown summary of the new concert

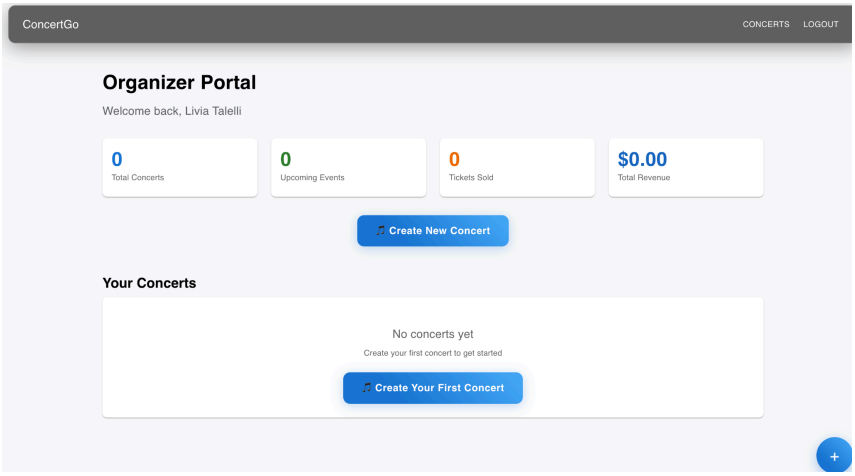
Outcomes:

- A new concert is successfully created and stored in the system
- The concert becomes visible in the organizer's dashboard
- The concert appears in the analytics report
- Fans will now be able to view and purchase tickets for the newly created concert
- System data is updated to reflect the concert's scheduling, resource usage, and availability for fan access

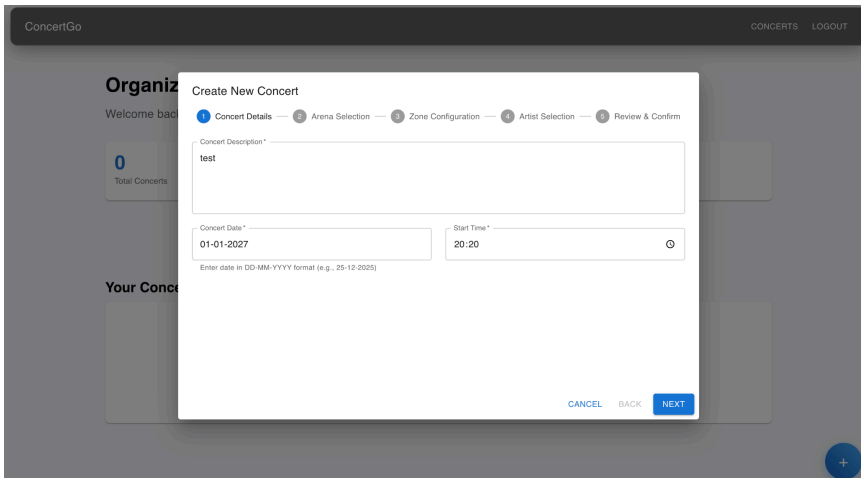
---

Steps:

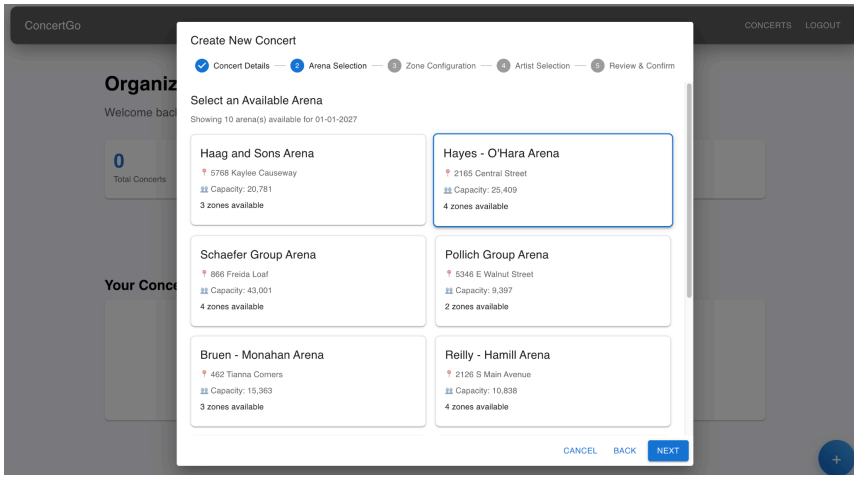
1- Click "Create New Concert" button



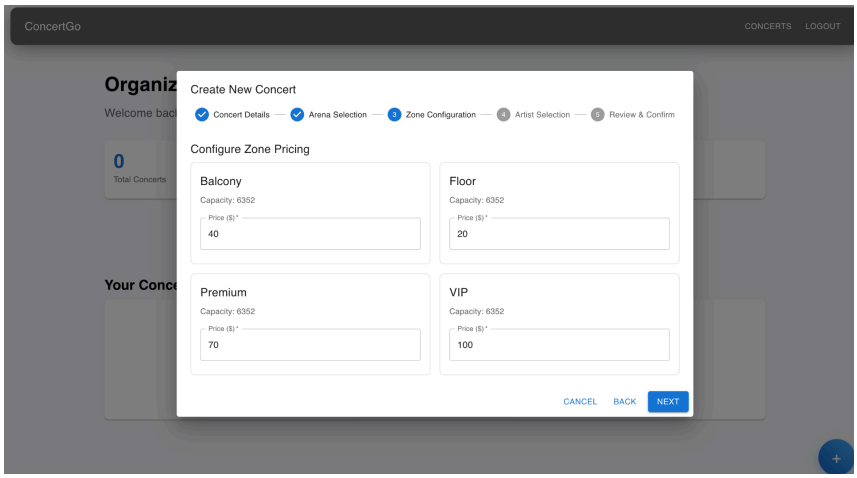
2- Enter concert details (description, date, and start time)



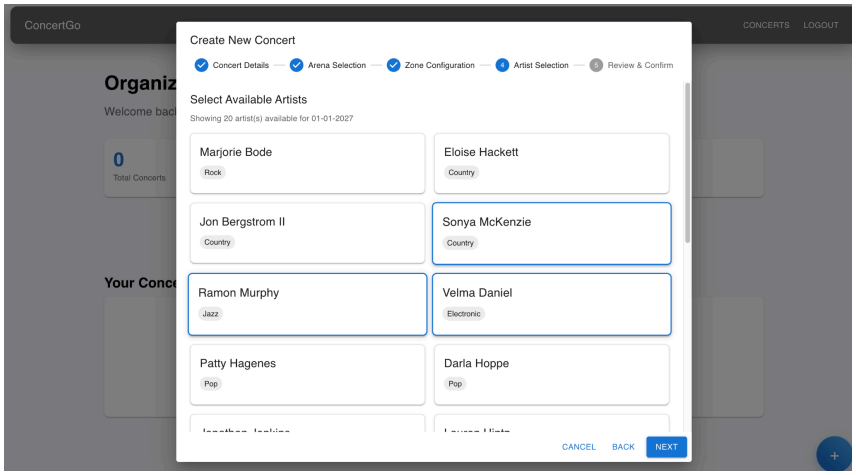
3- Selects an available arena



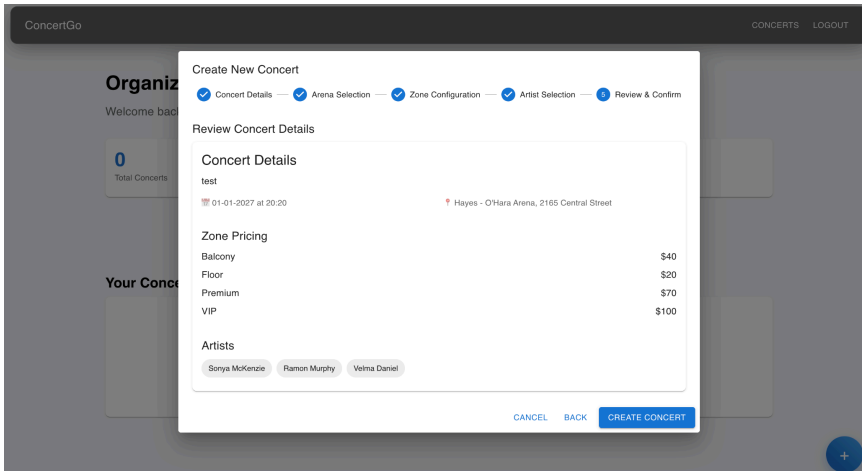
4- Enter pricing for every available zone



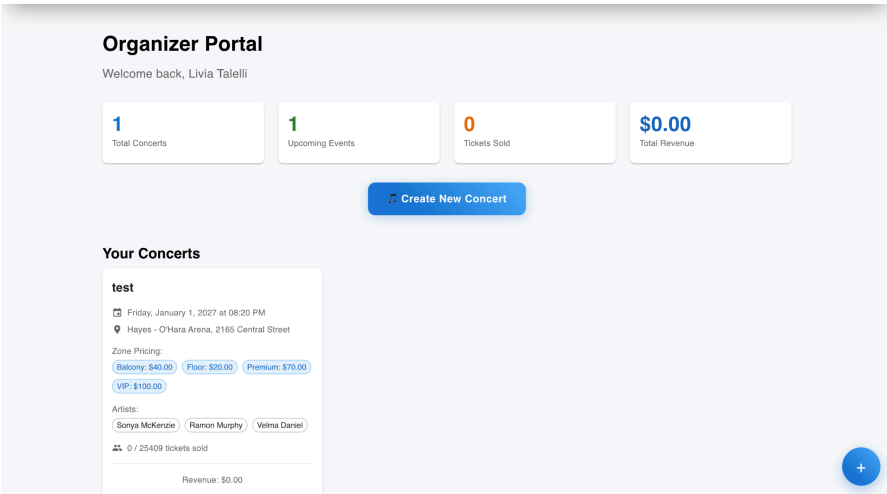
5- Selects one or more participating artist



6- Reviews all entered data and confirms the concert setup



7- View the new concert



Analytics Report: Organizer Concert Activity

Goal: To provide insights into organizer activity by tracking the number of concerts hosted in each arena. The report presents key metrics such as:

- **Name** and **contact details** of each organizer.
- The **arena name** where the concert is held.
- **Location** of the arena.
- Total number of concerts organized in each specific arena.
- Total **revenue** from ticket sales for concerts.
- Total count of **tickets purchased** per concert.
- Count of **unique fans** who purchased tickets for concerts in a given arena.
- **Average Ticket Price** calculated by dividing total revenue by the number of tickets sold.
- **Average Tickets per Concert ,** total tickets sold divided by the number of concerts organized.

This information helps the **superuser** assess which arenas are most utilized, identify active organizers, and spot patterns in event frequency across locations. It also supports strategic decisions for arena planning, resource allocation, and targeted outreach.

Filter Option: The report includes a filter to view concerts by arena name, enabling focused analysis per venue.

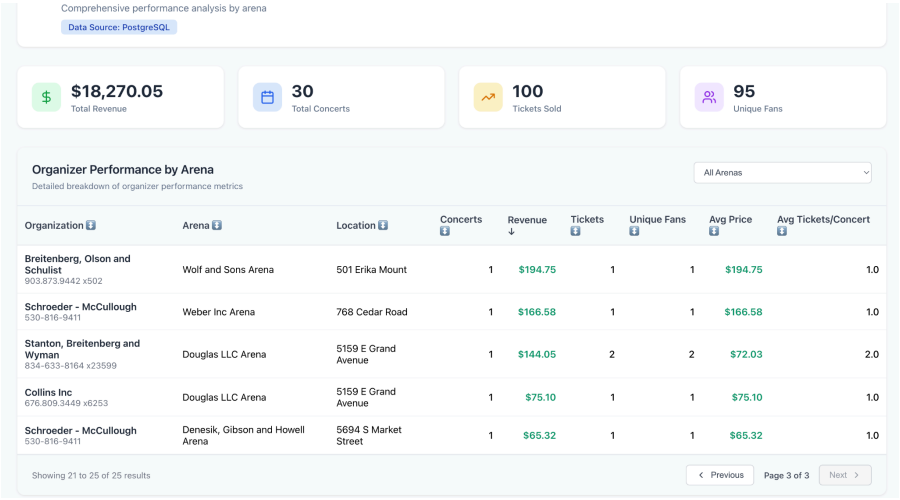
Outcomes (After Use Case Execution):

- When a new concert is created by an organizer, the total count of concerts per arena is updated in the report.

- Organizer and concert details are stored and reflected in the analytics table.
- The superuser can now:
    - Monitor arena usage trends
    - Evaluate organizer participation
    - Identify preferred arenas for future events
    - Make data-driven decisions for operational planning and marketing

---

Analytics Report Before Use Case has been performed:



Comprehensive performance analysis by arena

Data Source: PostgreSQL

$18,270.05 Total Revenue

30 Total Concerts

100 Tickets Sold

95 Unique Fans

Organizer Performance by Arena

Detailed breakdown of organizer performance metrics

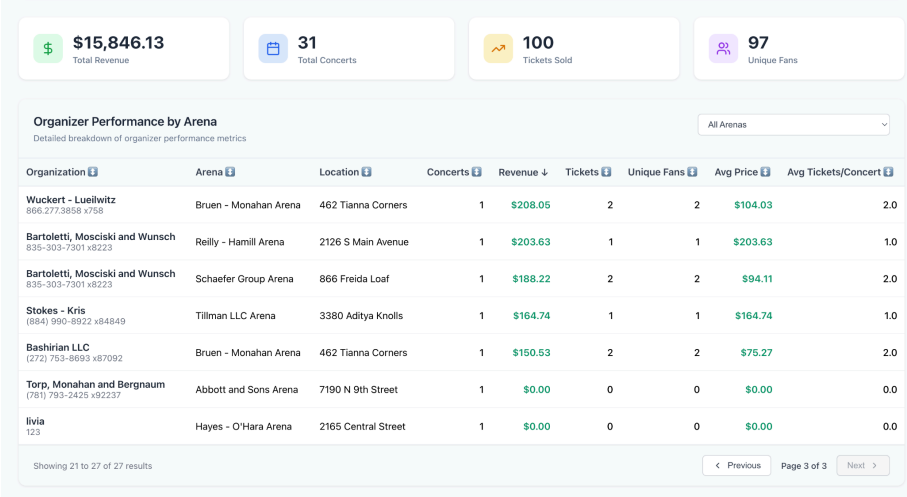
All Arenas

Organization	Arena	Location	Concerts	Revenue	Tickets	Unique Fans	Avg Price	Avg Tickets/Concert
Breitenberg, Olson and Schulist 903.873.9442 x502	Wolf and Sons Arena	501 Erika Mount	1	$194.75	1	1	$194.75	1.0
Schroeder - McCullough 530-816-9411	Weber Inc Arena	768 Cedar Road	1	$166.58	1	1	$166.58	1.0
Stanton, Breitenberg and Wyman 834-633-8164 x23599	Douglas LLC Arena	5159 E Grand Avenue	1	$144.05	2	2	$72.03	2.0
Collins Inc 676.809.3449 x6253	Douglas LLC Arena	5159 E Grand Avenue	1	$75.10	1	1	$75.10	1.0
Schroeder - McCullough 530-816-9411	Denesik, Gibson and Howell Arena	5694 S Market Street	1	$65.32	1	1	$65.32	1.0

Showing 21 to 25 of 25 results

Previous Page 3 of 3 Next

Analytics Report After Use Case has been performed:



$15,846.13 Total Revenue

31 Total Concerts

100 Tickets Sold

97 Unique Fans

Organizer Performance by Arena

Detailed breakdown of organizer performance metrics

All Arenas

Organization	Arena	Location	Concerts	Revenue	Tickets	Unique Fans	Avg Price	Avg Tickets/Concert
Wuckert - Lueilwitz 866.277.3858 x758	Bruen - Monahan Arena	462 Tianna Corners	1	$208.05	2	2	$104.03	2.0
Bartoletti, Mosciski and Wunsch 835-303-7301 x8223	Reilly - Hamill Arena	2126 S Main Avenue	1	$203.63	1	1	$203.63	1.0
Bartoletti, Mosciski and Wunsch 835-303-7301 x8223	Schaefer Group Arena	866 Freida Loaf	1	$188.22	2	2	$94.11	2.0
Stokes - Kris (884) 990-8922 x84849	Tillman LLC Arena	3380 Aditya Knolls	1	$164.74	1	1	$164.74	1.0
Bashirian LLC (272) 753-8693 x87092	Bruen - Monahan Arena	462 Tianna Corners	1	$150.53	2	2	$75.27	2.0
Torp, Monahan and Bergnaum (781) 793-2425 x92237	Abbott and Sons Arena	7190 N 9th Street	1	$0.00	0	0	$0.00	0.0
livia 123	Hayes - O'Hara Arena	2165 Central Street	1	$0.00	0	0	$0.00	0.0

Showing 21 to 27 of 27 results

Previous Page 3 of 3 Next

2.3 NoSQL Implementation

2.3.2 NoSQL Re-Design

Users Collection

```
{
  _id: string, // user_id from SQL
  email: string,
  user_password: string,
  first_name: string,
  last_name: string,
  registration_date: Date,
  last_login?: Date,
  user_type: 'fan' | 'organizer' | 'admin',

  // Fan-specific fields (if user_type === 'fan')
  fan_details?: {
    username: string,
    preferred_genre: string,
    phone_number: string,
    referral_code: string,
    referred_by?: string,
    referral_points: number,
    referral_code_used: boolean
  },

  // Organizer-specific fields (if user_type === 'organizer')
  organizer_details?: {
    organization_name: string,
    contact_info: string
  }
}
```

Artists Collection

```
{
  _id: string,
  artist_name: string,
  genre: string
}
```

Arenas Collection

```
{
  _id: string,
  arena_name: string,
  arena_location: string,
  total_capacity: number,
  zones: [
    {
      zone_name: string,
      capacity_per_zone: number
    }
  ]
}
```

Concerts Collection

```
{
  _id: string,
  concert_date: Date,
  time: string,
  description: string,
  organizer_id: string,
  arena_id: string,
  artists: [
    {
      artist_id: string,
      artist_name: string,
      genre: string
    }
  ],
  zone_pricing: [
    {
      zone_name: string,
      price: number
    }
  ]
}
```

Tickets Collection

```
{
  _id: string,
  fan_id: string,
  concert_id: string,
  arena_id: string,
  zone_name: string,
  purchase_date: Date,
  purchase_price: number,
  concert_date: Date,
  fan_username?: string,
  // Optional denormalized fields for analytics
  concert_name?: string,
  concert_time?: string,
  arena_name?: string,
  arena_location?: string
}
```

1. Users Collection

Current Decision: Embed Fan/Organizer/Admin Details

- **Embedding Benefits**: The 1:1 relationship between users and their specific role details (fan_details, organizer_details) ensures that authentication and user profile retrieval are handled without additional lookups, thus reducing query complexity and eliminating the need for joins. User authentication is a critical path operation that benefits significantly from single-document access.
- **Alternative (Using References)**:
    - **Benefits**: Easier to scale and modify fan/organizer data independently, removing data duplication and allowing for more flexible role-based access control.
    - **Drawbacks**: More joins are required for every user operation, increasing latency for authentication and profile management. The user_type field would need constant validation against separate collections.

2. Arenas Collection

Current Decision: Embed Zone Information

- **Embedding Benefits**: Zone configurations (zone_name, capacity_per_zone) are rarely updated once an arena is established, making them ideal for embedding. Seat availability calculations and zone-based pricing queries are significantly faster with embedded zone data. There is consistency in this option since there's no risk of zone data being stale between collections.

- **Alternative (Using References):**
    - **Benefits**: Zone modifications can be managed independently without locking the entire arena document. Avoids duplication if multiple arenas share similar zone structures.
    - **Drawbacks**: Requires additional queries for zone details during ticket purchasing and capacity calculations. Multiple joins are needed for concert-arena-zone operations.

3. Concerts Collection

Current Decision: Embed Artist and Zone Pricing Details

- **Embedding Benefits**:
    - Artist lineup and zone pricing for a concert rarely change once announced, making embedding suitable. Querying for available concerts with full artist information is faster without requiring joins. Direct access to embedded pricing details improves ticket purchasing use case performance significantly.
- **Alternative (Using References)**:
    - **Benefits**: Changes to artist information or pricing strategies propagate instantly without modifying concert documents. Better normalization reduces data redundancy.
    - **Drawbacks**: Overhead from join operations for every concert listing. Concert search by genre requires multiple collection queries instead of a single embedded field search.

4. Tickets Collection

Current Decision: Embed Denormalized Concert, Arena, and Fan Details

- **Embedding Benefits**:
    - Aggregations for fan behavior analysis, revenue reporting, and genre performance analytics are significantly faster without joins. Ticket details, once purchased, represent a historical snapshot that shouldn't change, making embedding ideal. This approach simplifies complex reporting queries and analytics use cases.
- **Alternative (Using References)**:
    - **Benefits**: Changes to concert, arena, or fan details don't require altering existing ticket documents. Better storage efficiency and reduced data duplication.
    - **Drawbacks**: Reporting use cases require multiple joins across collections, increasing latency. Aggregating ticket sales data across referenced collections is substantially slower than querying embedded data.

Also Indexing is set up for mongoDB

```
// User indexes - optimized for authentication and fan lookups
db.users.createIndex({ email: 1 }, { unique: true });
db.users.createIndex({ user_type: 1 });
db.users.createIndex({ 'fan_details.username': 1 }, { sparse: true, unique:
true });
db.users.createIndex({ 'fan_details.preferred_genre': 1 });
db.users.createIndex({ 'fan_details.referral_code': 1 }, { sparse: true,
unique: true });

// Artist indexes - optimized for genre searches
db.artists.createIndex({ genre: 1 });
db.artists.createIndex({ artist_name: 1 });

// Arena indexes - optimized for location searches
db.arenas.createIndex({ arena_location: 1 });

// Concert indexes - optimized for date and location searches
db.concerts.createIndex({ concert_date: 1 });
db.concerts.createIndex({ organizer_id: 1 });
db.concerts.createIndex({ arena_id: 1 });
db.concerts.createIndex({ 'artists.genre': 1 });
db.concerts.createIndex({ concert_date: 1, arena_id: 1 });

// Ticket indexes - optimized for fan queries and analytics
db.tickets.createIndex({ fan_id: 1 });
db.tickets.createIndex({ concert_id: 1 });
db.tickets.createIndex({ purchase_date: 1 });
db.tickets.createIndex({ concert_date: 1 });
db.tickets.createIndex({ fan_id: 1, concert_date: 1 });
db.tickets.createIndex({ concert_id: 1, zone_name: 1 });
```

2.3.3 NoSQL DB Setup and Data Migration

Database Setup

Dual Database Support:

The system supports both PostgreSQL (relational) and MongoDB (NoSQL).

The application distinguishes whether to use MongoDB or PostgreSQL through a configuration and **`migration status check`**. At runtime, a `migration status service` or configuration setting indicates which database is currently active or preferred.

When a service or controller needs to perform a data operation, it requests the appropriate repository from a repository factory. The factory consults the migration status and returns either a MongoDB or PostgreSQL repository implementation.

This dynamic selection allows the application to seamlessly switch between databases, ensuring that all data access operations are routed to the correct backend without requiring changes to the business logic or API layer.

MongoDB Schema & Indexing:

1. Collections for users, artists, arenas, concerts, and tickets are defined in `mongodb-schemas.ts` as Typescript Interfaces.
2. **Indexes** are created for common query patterns (e.g., unique email, username, referral code, and various search/filter fields).
3. **Connection Management:** A singleton connection manager (`mongoManager`) is used for MongoDB to ensure efficient and safe connection reuse.

Data Migration Process

Migration Service:

The migration logic is implemented in migration-service.ts. Migration is performed collection-by-collection (users, artists, arenas, concerts, tickets).

Step-by-Step Migration:

1. **Check Table Existence:** For each entity (e.g., users, tickets), the migration script first checks if the corresponding SQL table exists.
2. **Fetch Data from PostgreSQL**: Data is queried from the SQL tables, often **joining** related tables to gather all necessary fields (e.g., joining fans to users to get usernames).
3. **Transform Data:** Data is mapped to the MongoDB schema, including denormalized fields
4. **Insert into MongoDB:** The transformed data is inserted into the appropriate MongoDB collection using insertMany.
5. **Handle Special Cases:** Admin users, organizers, and fans are handled according to their specific fields.
6. Denormalized fields are populated to optimize future queries.

7. Index Creation: After migration, indexes are created on MongoDB collections to ensure efficient querying and data integrity.

Student 1 Jaupi, Joana (12448983)

2.3.4 NoSQL Use Case and Analytics Report

Brief Summary and Comparison with SQL Implementation

I implemented the fan ticket purchase use case using MongoDB, mirroring the logic previously developed in SQL. In this system, fans can buy tickets for concerts and apply discounts using either points or a referral code. The analytics report focuses on determining the number of tickets sold for upcoming concerts and the total revenue generated.

NoSQL Implementation:

Data Model:

Users collection stores fan information, including points and referral codes.

Concerts collection stores concert details (date, arena, etc.).

Tickets collection records each ticket purchase, referencing the user and concert, and storing price, discount, and payment details.

Purchase Logic:

When a fan buys a ticket, the system checks for available points or a valid referral code, applies the discount, and records the transaction in the Tickets collection.Points are deducted or referral codes are marked as used as appropriate.

Comparison with SQL Implementation (2.2):

In SQL, the logic relied on JOINs between normalized tables (users, concerts, tickets, referrals).

The NoSQL approach offers more flexibility for evolving requirements and can scale horizontally, but requires careful design to avoid data duplication and maintain consistency.

2.3.5 NoSQL Analytics Report Statement

To generate the analytics report (tickets sold and revenue for upcoming concerts), I used the following aggregation pipeline in MongoDB:

```
db.tickets.aggregate([
  { $match: { concert_date: { $gte: new Date() } } },
  { $lookup: {
      from: 'concerts',
      localField: 'concert_id',
      foreignField: '_id',
      as: 'concert_details'
  }},
  { $unwind: '$concert_details' },
  { $group: {
      _id: '$concert_id',
      concert_name: { $first: '$concert_name' },
      concert_date: { $first: '$concert_date' },
      description: { $first: '$concert_name' },
      arena_name: { $first: '$arena_name' },
      artists: { $first: '$concert_details.artists' },
      tickets_sold: { $sum: 1 },
      total_revenue: { $sum: { $ifNull: ['$purchase_price', 0] } }
  }},
  { $project: {
      _id: 0,
      concert_id: '$_id',
      concert_name: 1,
      concert_date: { $dateToString: { format: "%Y-%m-%d", date:
"$concert_date" } },
      description: '$concert_name',
      arena_name: 1,
      tickets_sold: 1,
      total_revenue: 1,
      artists: {
        $cond: [
          { $isArray: "$artists" },
          {
            $reduce: {
              input: "$artists",
              initialValue: "",
              in: {
                $cond: [
                  { $eq: ["$$value", ""] },
```

```
                  { $ifNull: ["$$this.artist_name", ""] },
                  { $concat: ["$$value", ", ", { $ifNull:
["$$this.artist_name", ""] }] }
                ]
              }
            }
          },
          ""
        ]
      }
  }},
  { $sort: { tickets_sold: -1 } }
])
```

The pipeline:

- Filters for upcoming concerts.
- Joins ticket data with concert details (to retrieve concert's artists)
- Groups by concert to count tickets sold and sum revenue.
- Formats the output for reporting

Design Impact on Execution

1. **Use of references and $lookup:**
   - The design keeps concert details in a separate concerts collection and uses references (`concert_id`) in the tickets collection.
   - The $lookup stage is used to join ticket data with concert details, which is similar to a SQL JOIN.
   - This approach keeps data normalized, making updates to concert details easier and consistent across all tickets.
2. **Denormalized Fields:**
   - Some fields (like concert_name, arena_name) may be denormalized in the tickets collection for performance, reducing the need for frequent lookups.
   - Denormalization can speed up reporting but risks data inconsistency if concert details change.
3. **Indexing:**
   - Indexes on concert_id and concert_date are critical for performance.
   - Without indexes, MongoDB would need to scan all documents, making the report slow as data grows.
   - With proper indexes, the aggregation pipeline can quickly filter and group relevant documents, as shown by reduced totalDocsExamined and executionTimeMillis in `.explain("executionStats")`.

Compromises and Mitigations

Compromise 1:

$lookup Performance

- Issue: MongoDB's $lookup is not as fast as SQL JOINs, especially on large collections.

Mitigation:

- Indexes on concert_id in tickets and _id in concerts collections.
- Denormalizing frequently accessed fields (like concert_name) in the tickets collection to avoid unnecessary lookups.

Compromise 2: Data Duplication vs. Consistency

- Issue: Embedding concert details in every ticket (full denormalization) would make reporting fast but lead to data duplication and **potential inconsistency if concert details change.**
- Mitigation:
    - Use references for concert details and only **denormalize non-volatile fields.**
    - Update logic ensures that only stable fields are denormalized, while **dynamic** details are always fetched via $lookup.

Design Decisions.

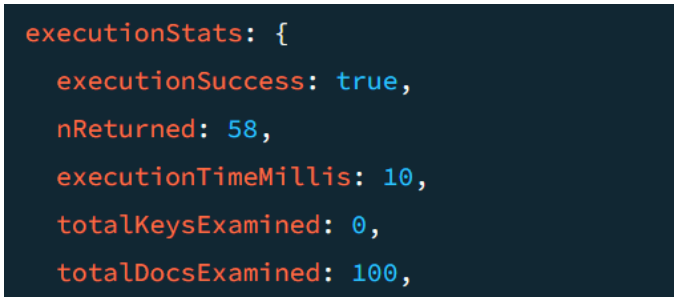
Selective Denormalization: Only non-volatile fields are denormalized in the tickets collection for performance.

Indexing: Indexes are created to optimize query performance for analytics.

2.3.6 NoSQL Indexing

To optimize the analytics report, I created an index on the `concert_date` field in the tickets collection. Running the aggregation pipeline with .explain("executionStats") shows that the query efficiently uses the `concert_date_1` index for filtering upcoming concerts. The output indicates that only 58 documents were examined to return 58 results, demonstrating a 1:1 scan-to-result ratio. The `$lookup` stage, which joins tickets to concerts to retrieve artist information, uses the default `_id_` index on the concerts collection. Confirming that the indexes significantly improve query performance. This efficient use of indexes ensures that the analytics report remains fast, even as the dataset grows.

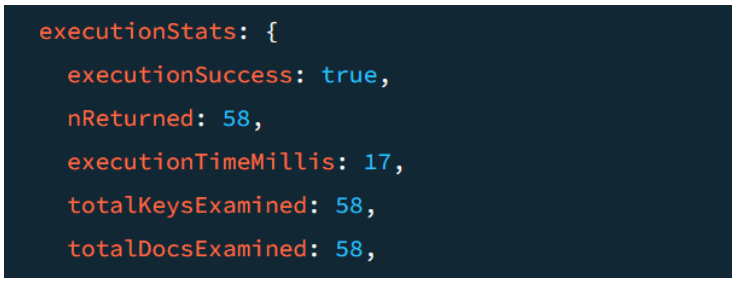
Before indexing:

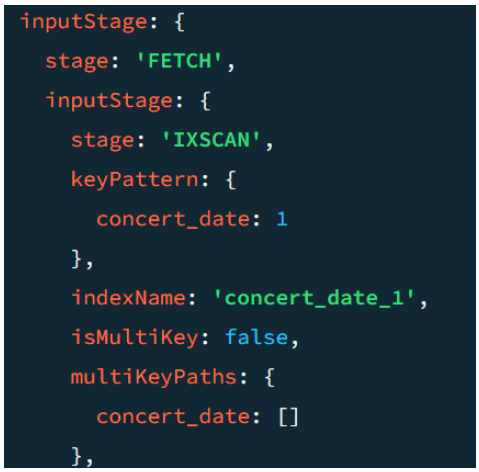


I then created the following indexes to test performance:

```
> db.tickets.createIndex({ concert_date: 1 })
```

After creating this index, I re-ran the analytics query with .explain("executionStats"). The execution stats showed a significant reduction in `totalDocsExamined`, and the indexes used field confirmed that the new index were being utilized.





Student 2 Talelli, Livia, 12449604

2.3.3 NoSQL Use Case and Analytics Report

Concert Creation Flow:

1. Organizer logs in and initiates concert creation.
2. Concert details are entered:
    - Description, date, time, arena, zones (with pricing), and artists.
3. Backend filtering for availability:
    - Arenas: The backend queries the concerts collection for concerts on the selected date and excludes arenas already booked.

    - Artists: Similarly, it queries for artists already performing on that date and excludes them.
4. Concert document structure:
    - When the organizer submits, the backend creates a single document in the concerts collection:

```
 {
"_id": "concert123",
"description": "Rock Night",
"concert_date": "2025-07-01T20:00:00Z",
"time": "20:00",
"organizer_id": "org456",
"arena_id": "arena789",
"zone_pricing": [
{ "zone_name": "VIP", "price": 100 },
{ "zone_name": "General", "price": 50 }
],
"artists": [
{ "artist_id": "art1", "artist_name": "Band A", "genre": "Rock" }
]
}
```

Zones and artists are embedded as arrays for fast access and reporting.

Resource Filtering:

Arenas and artists are filtered in real-time by querying the concerts collection for existing bookings on the selected date. This ensures no double-booking and up-to-date availability.

Outcome:

- The new concert is immediately available in the organizer's dashboard and for ticket sales.
- The concert document is indexed and can be efficiently queried for analytics

Comparison with SQL Implementation (2.2):

- **In SQL (PostgreSQL):**
- The logic relied on multiple JOINs between normalized tables (concerts, arenas, zones, artists, tickets).
- Concert creation required inserting into several tables and managing relationships with foreign keys.
- Analytics (e.g., concerts per arena, revenue) used GROUP BY and JOINs for reporting.

- The schema was strict, and changes required migrations.

- **In NoSQL (MongoDB):**
- All concert details (including zones and artists) are stored together in a single document in the concerts collection.
- Concert creation is a single insert operation, making it faster and more flexible.
- Analytics and resource filtering use aggregation pipelines and $lookup for joins.
- The schema is flexible and can evolve easily, but requires careful design to avoid data duplication and ensure consistency.

2.3.4 NoSQL Analytics Report Statement

Goal:

Provide real-time insights into organizer and arena activity, including:

- Number of concerts per arena
- Total revenue and ticket sales
- Unique fans per arena
- Average ticket price and tickets per concert

MongoDB Aggregation Pipeline:

```
db.concerts.aggregate([

{ $match: { arena_id: "0a3459c9-a78e-4d19-b9f1-68f2d1c6519c" } },

{ $lookup: {

from: "tickets",

localField: "_id",

foreignField: "concert_id",

as: "tickets"

}},

{ $group: {
```

```
_id: "$arena_id",
totalConcerts: { $sum: 1 },
totalRevenue: { $sum: { $sum: "$tickets.purchase_price" } },
totalTickets: { $sum: { $size: "$tickets" } },
uniqueFans: { $addToSet: "$tickets.fan_id" }
}},
{ $project: {
arena_id: "$_id",
totalConcerts: 1,
totalRevenue: 1,
totalTickets: 1,
uniqueFanCount: { $size: "$uniqueFans" },
avgTicketPrice: {
$cond: [
{ $eq: ["$totalTickets", 0] }, 0,
{ $divide: ["$totalRevenue", "$totalTickets"] }
]
},
avgTicketsPerConcert: {
$cond: [
{ $eq: ["$totalConcerts", 0] }, 0,
{ $divide: ["$totalTickets", "$totalConcerts"] }
]
```

```
}

}}

])
```

The aggregation pipeline consists of the following stages, each serving a specific purpose:

- Filters concerts by organizer.
- The $match stage restricts the pipeline to concerts created by a specific organizer (using organizer_id), enabling index usage and reducing the number of documents processed.
- Joins ticket data with concert details.
- The $lookup stage joins each concert with its related tickets from the tickets collection, similar to a SQL JOIN.
- Groups by organizer to aggregate metrics.
- The $group stage calculates the total number of concerts, total revenue, total tickets sold, and unique fans for each organizer.
- Formats the output for reporting.
- The $project stage computes averages (e.g., average ticket price, average tickets per concert) and shapes the final output for the report.

Compromises and Design Decisions:

1. Data Modeling: Embedding vs. Referencing

Compromise: In MongoDB, we store concerts and tickets in separate collections, referencing concerts from tickets via concert_id. This is necessary because tickets can be numerous and are created after the concert document, making embedding impractical for scalability and write patterns.

Mitigation: We use referencing and $lookup to join ticket data at query time. This design allows for efficient writes and avoids document size limits, at the cost of more complex aggregation queries and potentially slower analytics compared to embedded models.

2. Aggregation: Performance: Use of $lookup

Compromise: The $lookup stage (join) can be expensive, especially as the number of concerts and tickets grows, since MongoDB is not as optimized for joins as SQL databases.

Mitigation: To minimize performance impact, we: Place a $match stage before $lookup to filter concerts by arena_id or organizer_id, reducing the number of documents that need to be joined. Ensure indexes exist on concerts.arena_id, concerts.organizer_id, and tickets.concert_id to speed up both the match and join operations.

3. Unique Fan Calculation:

Compromise: Calculating unique fans per arena requires collecting all fan_ids into an array with $addToSet and then counting them. This can be memory-intensive for arenas with many fans.

Mitigation: We use $addToSet and $size in the aggregation pipeline, which is efficient for moderate data sizes. For very large datasets, we could consider pre-aggregating unique fan counts periodically and storing them in a summary collection.

4. Indexing Trade-offs:

Compromise: Adding indexes improves read/query performance but can slow down writes and increase storage requirements.

Mitigation: We selectively index only the fields most relevant to analytics and filtering, balancing query speed with write performance and storage costs.

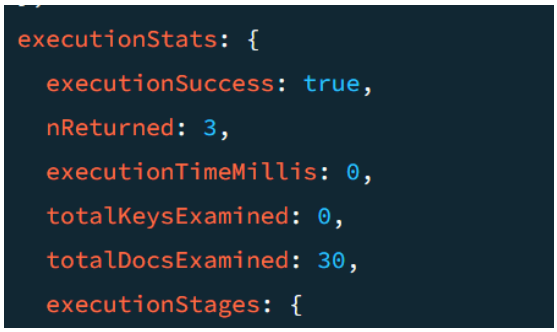
2.3.5 NoSQL Indexing

Indexes Used:

concerts.arena_id: To quickly find all concerts in a specific arena.

```
db.concerts.createIndex({ arena_id: 1 })
```

Before indexing:



After indexing:

- **Analysis:** As shown above, implementing indexes on key fields such as **arena_id**, led to a dramatic reduction in the number of documents MongoDB had to scan for analytics queries.
- In the output, index usage is indicated by the presence of an IXSCAN stage in the query plan and a nonzero totalKeysExamined value.
- **Query Performance:** With this index, aggregation pipelines and lookups are much faster.
- **Before indexes:** MongoDB performed a full collection scan, examining every concert document to find matches for analytics.
- **After indexes:** With indexes in place, MongoDB could “jump” directly to the relevant documents.