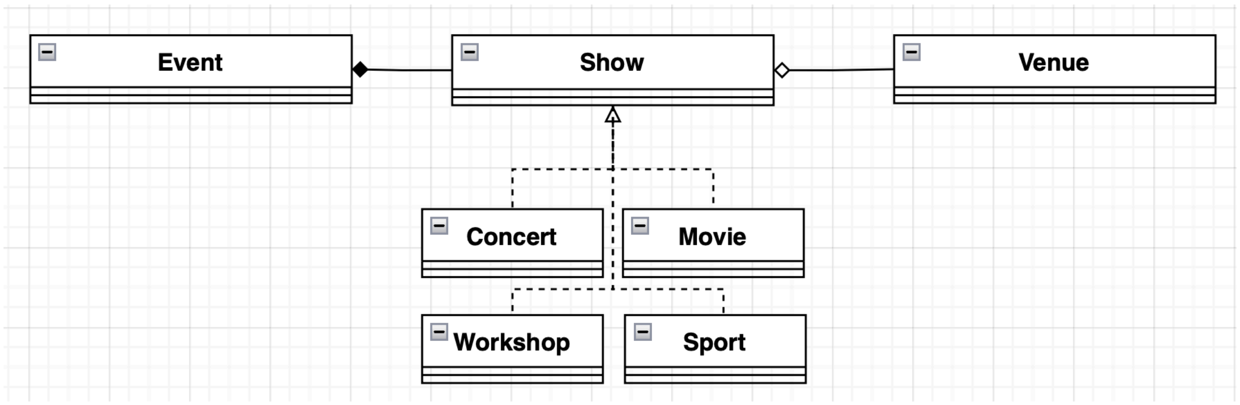
Ticket Booking App

Database Design:

1. Breaking down the whole db design into 4 major components
   a. Event Management
   b. Seat Management
   c. Booking Management
   d. User Management
2. Relations and Object identifications

   a. Event Management

   Events are like bigger plans. Like a movie releasing or a music tour
   Now each event is associated with *Shows*. Shows are like a movie show in a particular theatre or a music event in Bangalore on a particular date. Each *Show* will be performed at a *Venue.*



   b. Seat Management
   - *Venue Onboarding tables*
     Each venue can be onboarded on the platform.
     Each venue has a unique seat layout.
     The layout is broken into sections, with each section having some priority/ value
     These are a part of the venue onboarding process, and will usually remain static

   - *Show Launching tables:*
     With each new show launching, there will be pricing decided for each section. This will be a part of the launching of a new show.

   c. Booking Management
   - *Inventory*
     Once you decide to make your show live, there needs to be static data, which will be created to pre-populate all the available tickets.

- *Tickets, Payments*
  Once the payment confirmation is received, we add the transaction details to the Payments tables. Booking details are added to the Booking table and the individual tickets are logged in the tickets table

d. User Management
   All other tables related to users are managed separately. Visitors, Organisers, Venue Owners

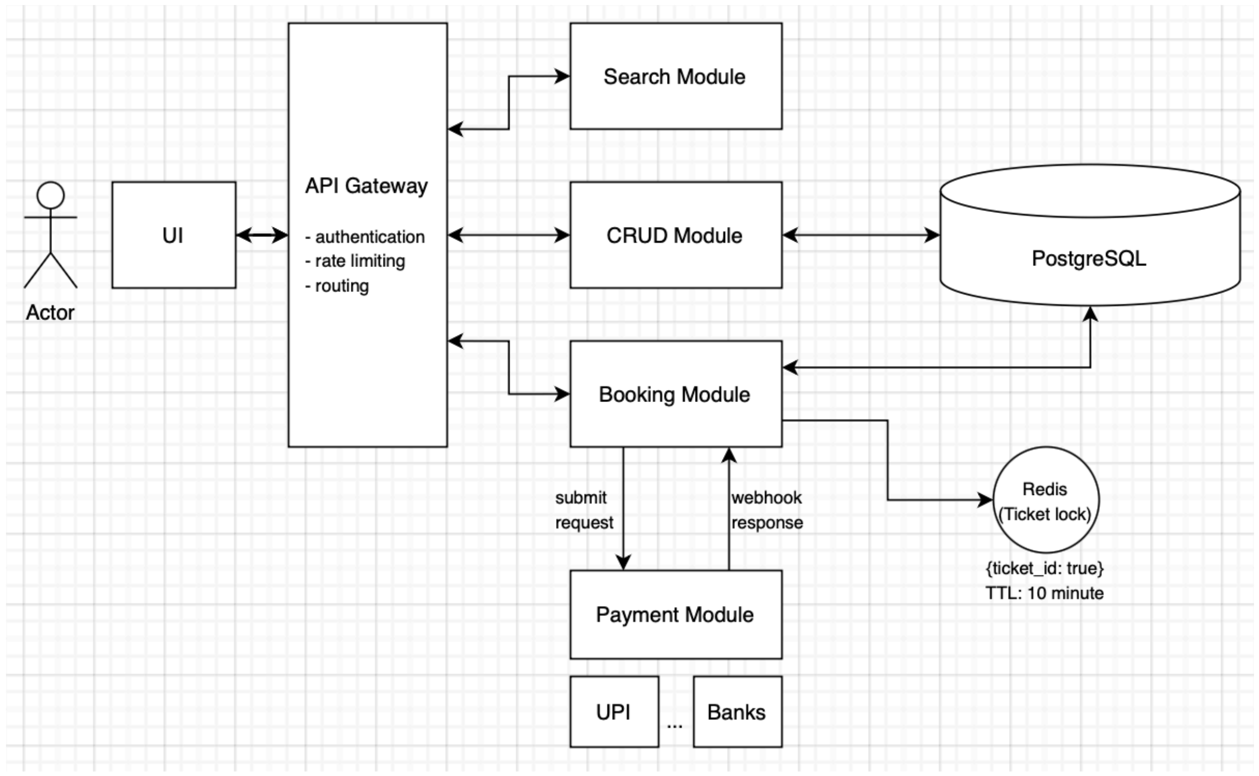
System Architecture:

We are planning on creating a system, which should

a. Booking should be very consistent
b. CRUD operations should be highly available
c. The DB is expected to be read heavy sometimes, to fetch available seats

As a part of this assignment I plan to explore on the three modules

a. CRUD operations: fetching the shows and the available seats
b. Search Module: searching for ideal results
c. Booking Module: making sure that system is fault tolerant even during sudden traffic



This architecture diagram could help us implement this product.
User using a mobile app or web app. All the requests reach the API gateway.

API gateway does user authentication, rate limiting and routes the request to the concerned module.
In our backend we can have a search module, basic data fetching module, and a booking module
All out configurations and data stays in the PostgreSQL. Choosing a SQL database because

a. Our data is very structured
b. We can use data joins to data efficiently
c. ACID properties to required to support consistency

While we are booking, once I go to the payment page, there should be a timer, let's say about 10min. We should not allow other users to book the same seats during this time period. There are multiple ways we could achieve this.

a. We could update the inventory table on the Pg to "reserved". But if the person lets the app, it becomes difficult to undo this
b. We can create a cron job, which runs every 15 min, and unreserve the seats that we reserved for more that 15 min. But this will results in some stale values and is not real time
c. Best approach: we could have a separate redis cache. For reservations we update this table with a TTL of 10 min. The cache will just be containing seat_value and true/ false as values

For the search module, I am thinking if we fire a search on the SQL db using LIKE operator, it might be slow since we will need to go through all the rows and fetch the best searches, instead we can have another nosql/ document db, which will be serving for the searches. Eg Elasticsearch.
The next challenge will be to keep it up-to-date/consistent with the information in the Pg.

a. For this we can add the details there as well when we add a new event. But might result in inconsistencies when one fails and another succeeds.
b. Best approach: we can configure logstash with a cron job, eg every 10 min. It will fetch the latest data from Pg, process it and add it to the Elastic. Light weight and efficient

This might also help us in building some ML models in future on this text data.

API Design:

I have not really used GraphQL, but from my understanding I know it makes more sense to use it when we are trying to create a read-heavy app. Where there are multiple things that need to be shown on the UI, and we do not want to make multiple API calls and then club all the results in the UI.

For us, we have a simple list of events that we need to show on the home, then an info page where we fetch the info about that show and then fetch the available bookings. I'll choose RESTAPI atleast for now, and if in future we need to we can think about migrations.

For bookings, we want to make sure that our system is idempotent and transactional in nature. For these properties, REST is the best option.

I have identified these 3 key endpoints.

1. *POST /api/v1/book/reserve*
   Request:
   Authentication: Required, via Bearer token
   Body: { "show_id": 123, "seat_ids": ["A1", "A2"]}
   Fields
   show_id *(number/int) (mandatory)*: Show identifier
   seat_ids *(array of string) (mandatory)*: Seat identifiers

   Success Response:
   - 202 Accepted

   Content: {"status": "accepted", "show_id": 123, "seat_ids": ["A1", "A2"]}

   Error Response:
   - 400 Bad Request

   {"error_code": "INVALID_REQUEST", "message": "seat_ids cannot be empty"}

   - 401 Unauthorized

   {"error_code": "UNAUTHORIZED", "message": "Authentication required"}

   - 403 Forbidden

   {"error_code": "FORBIDDEN", "message": "Not allowed to reserve seats"}

   - 409 Conflict

   {"error_code": "SEAT_LOCKED", "message": "One or more seats are already reserved", "details": {"show_id": 123, "locked_seats": ["A1"]}}

   - 500 Internal Server Error

   {"error_code": "INTERNAL_ERROR", "message": "Something went wrong"}

2. *POST /api/v1/book*
   Request:
   Authentication: Required, via Bearer token
   Idempotency-Key: Required, unique-uuid-key
   Body: { "show_id": 123, "seat_ids": ["A1", "A2"]}
   Fields:
   show_id *(number/int) (mandatory)*: Show identifier
   seat_ids *(array of string) (mandatory)*: Seat identifiers

   Response:
   - 201 Created

{"status": "seats_booked", "booking_id": 987, "transaction_id": "txn_8f3a2c7e-4d2b-4cda-9c62-7d6ad7b1a7d9", "show_id": 123, "seat_ids": ["A1", "A2"]}

- 200 OK (idempotent replay)

{"status": "seats_booked", "booking_id": 987, "transaction_id": "txn_8f3a2c7e-4d2b-4cda-9c62-7d6ad7b1a7d9", "show_id": 123, "seat_ids": ["A1", "A2"]}

Error Response:

- 400 Bad Request

{"error_code": "INVALID_REQUEST", "message": "seat_ids cannot be empty", "details": {}}

- 401 Unauthorized

{"error_code": "UNAUTHORIZED", "message": "Authentication required", "details": {}}

- 404 Not Found

{"error_code": "SHOW_NOT_FOUND", "message": "Show 123 not found", "details": { "show_id": 123 }}

- 409 Conflict

{"error_code": "SEAT_NOT_AVAILABLE", "message": "Some seats are not available", "details": {"show_id": 123,"unavailable_seats": ["A1"]}}

500 Internal Server Error
{"error_code": "INTERNAL_ERROR", "message": "Something went wrong"}

3. *GET /api/v1/shows?cat={category}&city={city}*

Request:
Authentication: Required, via Bearer token

Response:

- 200 OK

```
[
  {
    "show_id": 101,
    "event_id": 12,
    "category": "movie",
    "title": "Pushpa 2: The Rule",
    "start_time": "2026-01-10T18:30:00+05:30",
    "end_time": "2026-01-10T21:00:00+05:30",
    "venue_name": "PVR: Phoenix Marketcity",
    "city": "Bangalore",
    "min_price": 250,
    "currency": "INR"
  },
  {
    "show_id": 102,
    "event_id": 15,
    "category": "concert",
    "title": "Arijit Singh Live",
    "start_time": "2026-01-11T19:30:00+05:30",
    "end_time": "2026-01-11T22:00:00+05:30",
    "venue_name": "INOX: Orion Mall",
    "city": "Bangalore",
    "min_price": 450,
    "currency": "INR"
  }
]
```

Error Response:

- 400 Bad Request

```
{
  "error_code": "INVALID_REQUEST",
  "message": "category must be one of: movie, concert, all",
  "details": {
    "category": "sports"
  }
}
```

- 401 Unauthorized

```
{"error_code": "UNAUTHORIZED",  "message": "Authentication required", "details": {}}
```

- 403 Forbidden

```
{"error_code": "FORBIDDEN", "message": "Access to this resource is forbidden",
"details": {}}
```

- 500 Internal Server Error

```
{"error_code": "INTERNAL_ERROR", "message": "Something went wrong", "details": {}}
```

4. GET /show/{show_id}
5. GET /search?q={query}

Scalability and Performance:

For handling sudden huge traffic, e.g. for a Diljit concert ticket.
Bottleneck identification:

1. Fetching the real time available seats
2. Making sure that not all tickets gets reserved within few seconds, and then we have to slowly release reserved tickets if payment not done or payment failed

Solutions:

1. Basic solution should be to scale the crud servers, increase read replicas
2. For such cases we can enable some extra components in our design.
    a. We can add a virtual queue in the system, like a queue we have outside the ticket counter. And allow 10-20 people to enter the system slowly
    b. This will make sure that not all the users rush to the /booking endpoint at once, and we are able to serve everyone with proper up-to-date information
    c. This queue will be added after the API gateway and before the booking module
3. We can toggle this queue based on the shows, so that will be configurable and will be show specific.

PS. to keep it fair, we can also put a cap on the number of seats a single user can book at a time.

Caching Strategies:
For improving the performance, we can also add a CDN for the images and maybe the trailers that we plan to show in the info page before the API gateway.
It will cache based on the regions.
Also since its region wise cache, we can also have API responses cached there, for latest region wise events
But we will have to make sure that the seat availability APIs are never from cache, we need to make sure that we provide the latest results to the users for these endpoints

User Authentication and Authorization:

For now i am having saving the email_id and password hash in a table, and using that for user authentication
Once a user logs in, we check in this db for password hash.
Once authenticated, we check for the user roles in the roles db. user/organiser/venue_owener etc
Create a JWT token, with all this information and pass it back to the UI. This token expires in 15 min. And the UI is expected to make another call for the new token.
For refresh we will expose another endpoint /refresh_token

In future we can onboard different oauth2 providers like google, facebook etc or the OTP based verification for the user authentication, but the roles and scopes should be handled by us on our database.

Ideally this will be a completely different service, and all out other microservices will be connecting calling it for token creation and token verification.

For authorization, we can add the roles and scopes for the user in the token itself. But this will be for the putting up UI, and having the correct UI loading.
Eg, the organizer should be able to see the event management page.
Now for what all needs to be shown on this page, and API call will be made to the backend, we will check if the person is an organizer for that particular event. If yes, only then the information will be passed back

Once we reach the API, based on the username and role, we will check if the person is an attendee/organizer/venue_owner for that event, and a response will be sent accordingly.

Real-Time Data Handling:

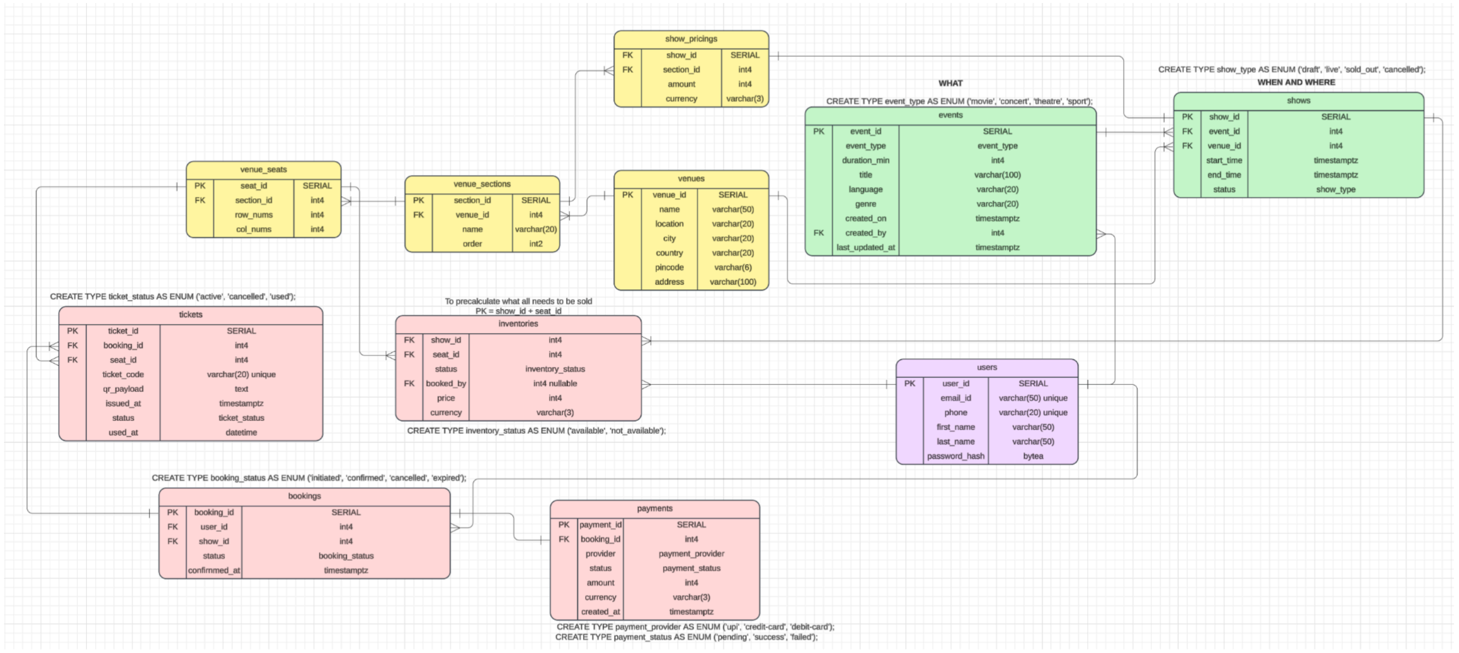
I am making sure that my seat availability source of truth is the "inventory" table.
It pre-pollulates all the seats that are for sale, and as the payment is processed. This table is updated as SOLD for all the seat rows along
With this we will also add a row in the "bookings" table with the booking id.
And split this booking into multiple rows per seat in the "tickets" tables, to create a different ticket id for each.

All 3 things are happening as a transaction asynchronously. One fails, everything fails.

Once this succeeds we show the users that the booking is done, and the tickets will be mailed to them

After this we will be maintaining an event queue as well. This booking will be added to the event queue. The subscribers to this queue can be ticket_generation service(taking care of creating a unique QR code), email notification service, mobile notification service, whatsapp notifications service etc

Databases:



show_pricings

FK	show_id	SERIAL
FK	section_id	int4
	amount	int4
	currency	varchar(3)

```
CREATE TYPE event_type AS ENUM ('movie', 'concert', 'theatre', 'sport');
```

events

PK	event_id	SERIAL
	event_type	event_type
	duration_min	int4
	title	varchar(100)
	language	varchar(20)
	genre	varchar(20)
	created_on	timestamptz
FK	created_by	int4
	last_updated_at	timestamptz

```
CREATE TYPE show_type AS ENUM ('draft', 'live', 'sold_out', 'cancelled');
```

shows

PK	show_id	SERIAL
FK	event_id	int4
FK	venue_id	int4
	start_time	timestamptz
	end_time	timestamptz
	status	show_type

venue_seats

PK	seat_id	SERIAL
FK	section_id	int4
	row_nums	int4
	col_nums	int4

venue_sections

PK	section_id	SERIAL
FK	venue_id	int4
	name	varchar(20)
	order	int2

venues

PK	venue_id	SERIAL
	name	varchar(50)
	location	varchar(20)
	city	varchar(20)
	country	varchar(20)
	pincode	varchar(6)
	address	varchar(100)

```
CREATE TYPE ticket_status AS ENUM ('active', 'cancelled', 'used');
```

tickets

PK	ticket_id	SERIAL
FK	booking_id	int4
FK	seat_id	int4
	ticket_code	varchar(20) unique
	qr_payload	text
	issued_at	timestamptz
	status	ticket_status
	used_at	datetime

inventories

FK	show_id	int4
FK	seat_id	int4
	status	inventory_status
FK	booked_by	int4 nullable
	price	int4
	currency	varchar(3)

```
CREATE TYPE inventory_status AS ENUM ('available', 'not_available');
```

users

PK	user_id	SERIAL
	email_id	varchar(50) unique
	phone	varchar(20) unique
	first_name	varchar(50)
	last_name	varchar(50)
	password_hash	bytea

```
CREATE TYPE booking_status AS ENUM ('initiated', 'confirmed', 'cancelled', 'expired');
```

bookings

PK	booking_id	SERIAL
FK	user_id	int4
FK	show_id	int4
	status	booking_status
	confirmed_at	timestamptz

payments

PK	payment_id	SERIAL
FK	booking_id	int4
	provider	payment_provider
	status	payment_status
	amount	int4
	currency	varchar(3)
	created_at	timestamptz

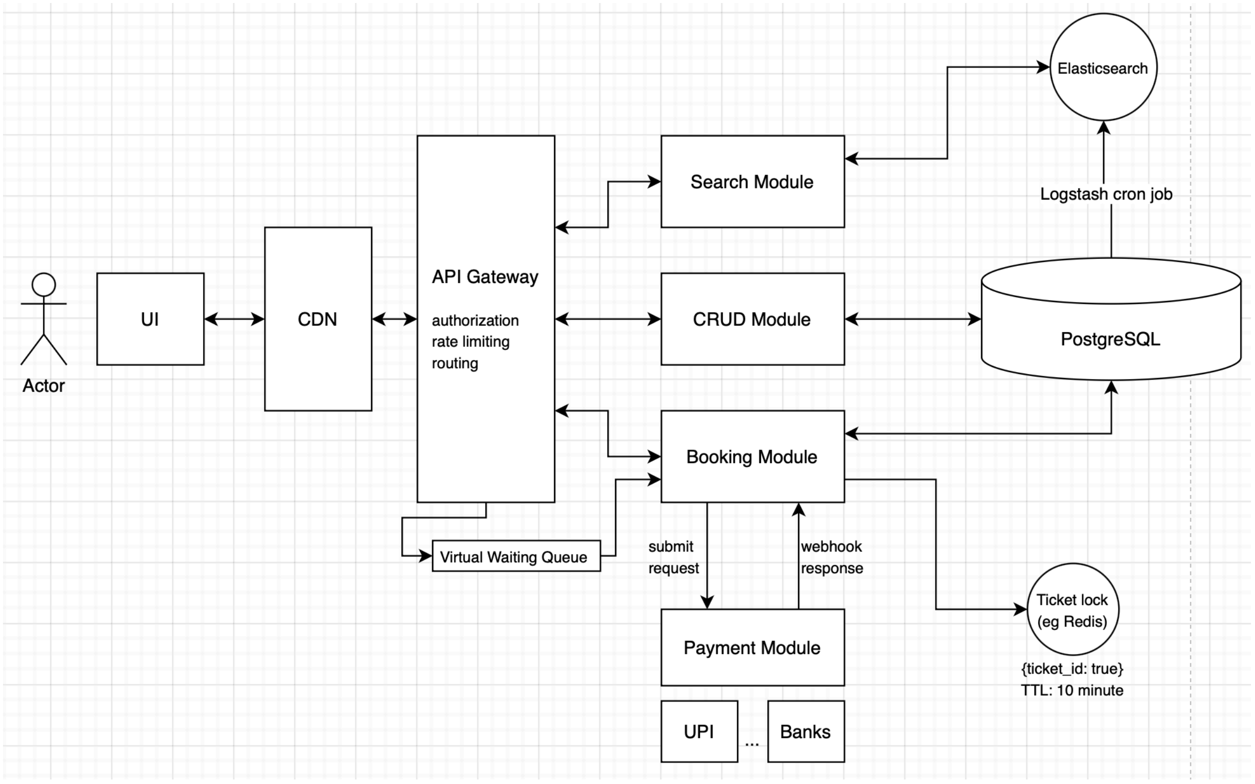
```
CREATE TYPE payment_provider AS ENUM ('upi', 'credit-card', 'debit-card');
CREATE TYPE payment_status AS ENUM ('pending', 'success', 'failed');
```

Attaching the Lucid and the SQL queries to generate this structure.

Some other DB hosting strategies that we can try are:

- For faster querying we can keep rotating the database. Have a hot and cold version. Where the hot versions are kept in RAM for faster reading. Older events are moved to the cold versions.
- We can shard the DB by the event locations, and host them in those regions.

The Final HLD:


Elasticsearch
Search Module
Logstash cron job
Actor
UI
CDN
API Gateway
authorization
rate limiting
routing
CRUD Module
PostgreSQL
Booking Module
Virtual Waiting Queue
submit
request
webhook
response
Ticket lock
(eg Redis)
{ticket_id: true}
TTL: 10 minute
Payment Module
UPI
...
Banks