Lottery Project

Functional Requirments & Assumptions

- **Only one event** can be created each day.
- **Users can register for lottery** events as participants.
- Lottery participants will be able to **buy as many lottery ballots as possible** which is not closed yet.
- Each day at **midnight the lottery event will be closed** and a **random lottery-winning ballot** will be selected from all the participants.
- All users will be able to check the winning ballot for any **specific date.**

Additional

- The winner will be notified via email.
- Users have to pay per lottery ballot submission.

Domain Model / Entities

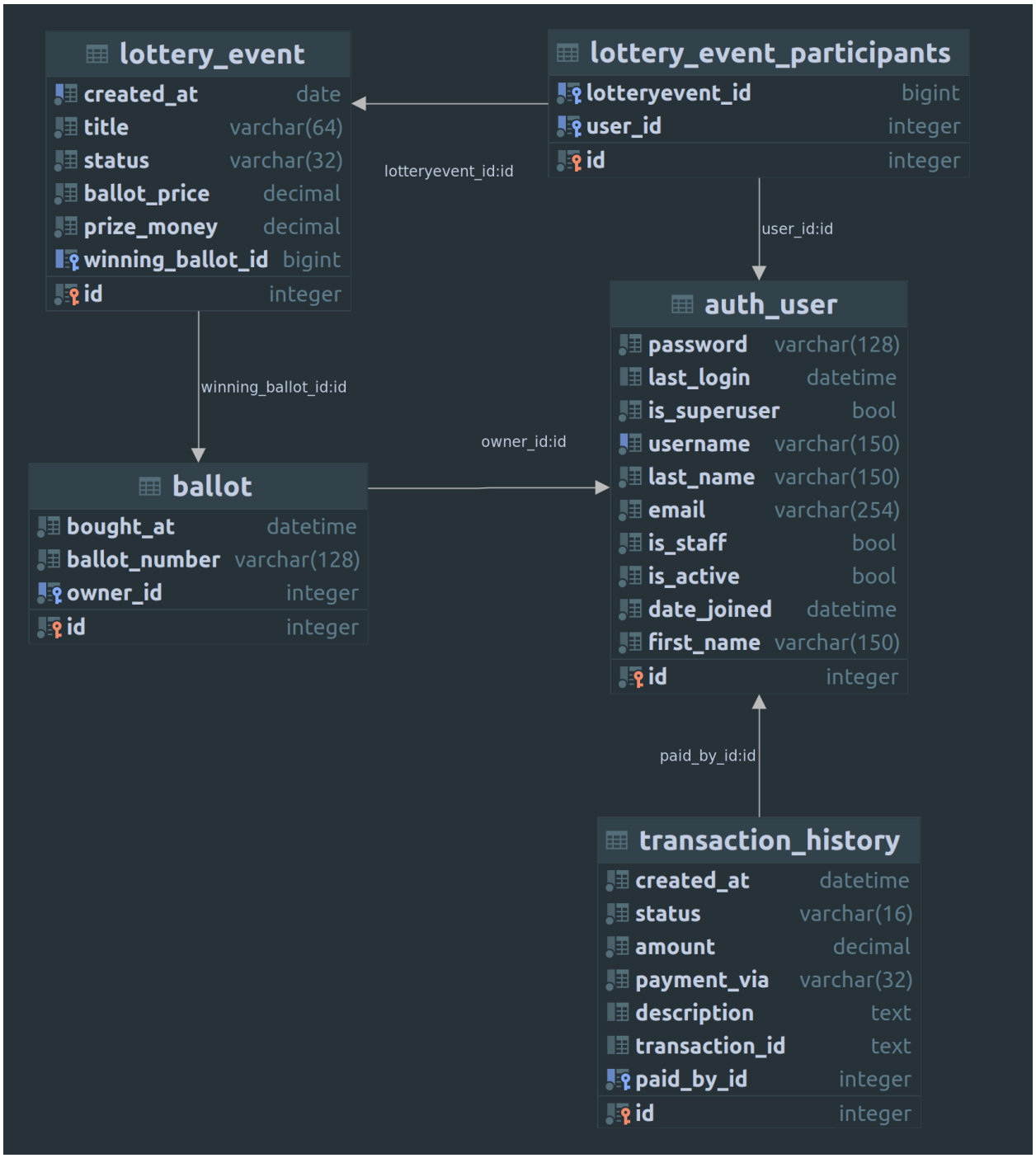
1. User
Used Django default User model class but for project purposes used only **Username** and **Email** and other fields are kept optional.

2. Lottery Event
Minimal information about a lottery event. The User model is connected to it as participants. Though the requirement says for one lottery event only, the system handles multiple lottery events on the same day.

3. Ballot
Minimal information about a ballot. The User model is connected as the owner of a ballot. Each lottery event is assumed to have only one winning ballot.

4. Transaction History
Minimal information about a transaction history after payment is made by the users.

ER Diagram


lottery_event
created_at date
title varchar(64)
status varchar(32)
ballot_price decimal
prize_money decimal
winning_ballot_id bigint
id integer
lottery_event_participants
lotteryevent_id bigint
user_id integer
id integer
lotteryevent_id:id
user_id:id
auth_user
password varchar(128)
last_login datetime
is_superuser bool
username varchar(150)
last_name varchar(150)
email varchar(254)
is_staff bool
is_active bool
date_joined datetime
first_name varchar(150)
id integer
winning_ballot_id:id
owner_id:id
ballot
bought_at datetime
ballot_number varchar(128)
owner_id integer
id integer
paid_by_id:id
transaction_history
created_at datetime
status varchar(16)
amount decimal
payment_via varchar(32)
description text
transaction_id text
paid_by_id integer
id integer

APIs and functionalities

User CRUD operations (list, create, details, update, delete)

Lottery event CRUD operations (list, create, details, update, delete)

Register user - Register a user for a particular lottery event.

Make Payment - Dummy API assuming a payment service is working in the backend and returns transaction information on success.

Purchase ballot - Get a ballot for a particular lottery event. This is expected to call after make payment is successful.

Close active lotteries - This endpoint is exposed to close and select the winner of all active lotteries immediately. There is a scheduler that triggers at 12.01 AM UTC time that performs this functionality as well.

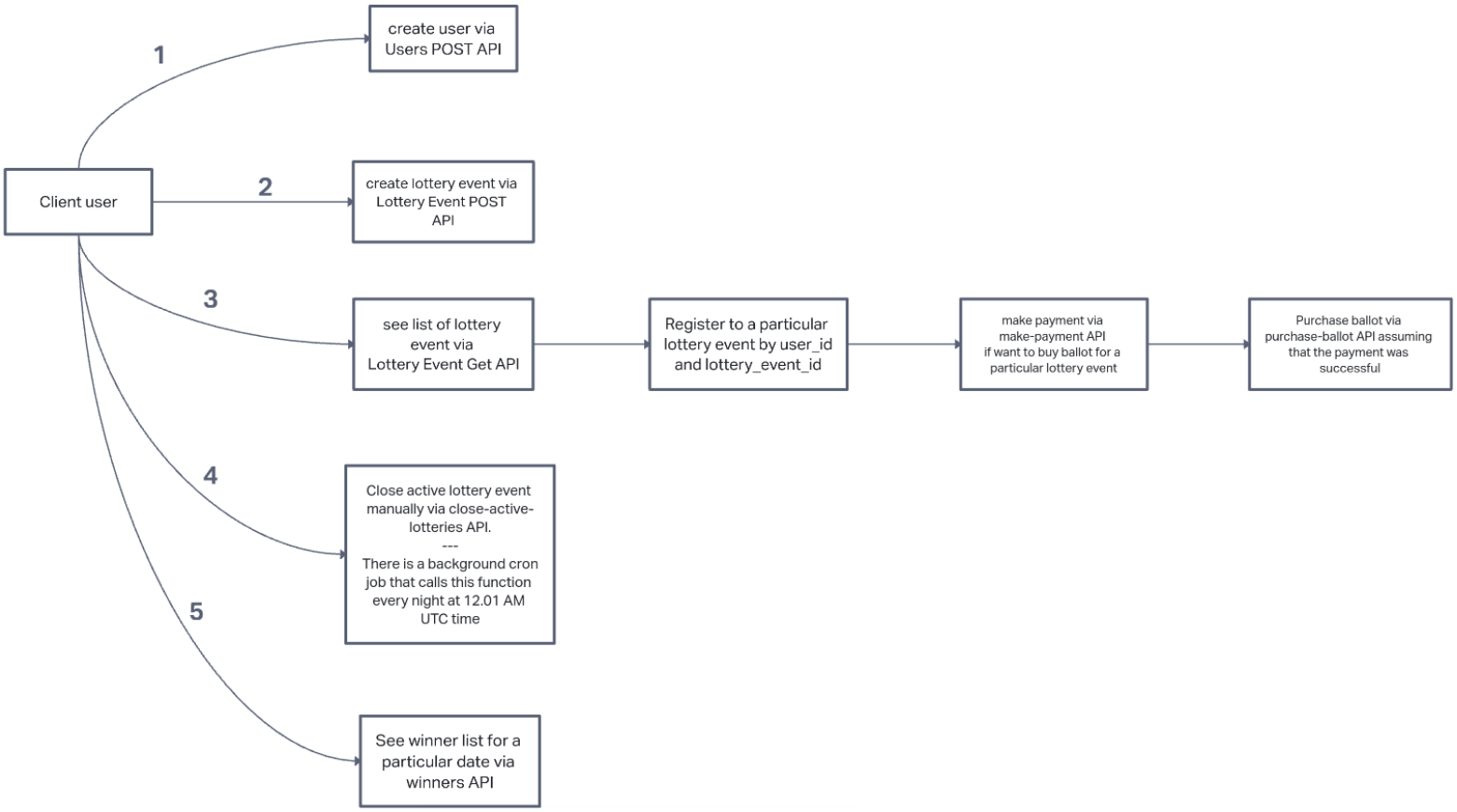
Show winners - Returns winning ballot, ballot winner for lottery events on a particular date.

API Token
hKhgiWl65LLRjtkRGDvDKZP5MokBtzLzz7mhUQr0DCgQvzXtFbx9fAxnClClQU6YnZ80v

Postman collection

Documentation

User Journey



Instructions to run the application

Go to the project root folder **lottery** and run the command

docker-compose up –build

The app will run on **0.0.0.0:8081**.

Things I'm glad to have done

1. Design decisions of model

The relationship between the **user** and the **lottery event** model is made **many to many** because both the user and the lottery event model can have multiple relationships with each other assuming that there can be multiple lottery events in future.

The relationship between the **user to ballot & transaction history** model is made **one as many** as a user can have multiple entries for both ballot and transaction history according to the requirement.

The relationship between the **lottery event and the ballot** is made **one-to-one** assuming that each lottery event can have only one winning ballot. If the requirement were **multiple winning ballots for a lottery** then it would be **one to many** relationships.

2. **Separation of concern**

The application codebase and file organization is structured in such a way which ensures a common structured base on entity object. Each entity has its own **views or controllers, serializers, tests, helpers, managers** etc and each of them has different **roles** and **responsibilities** within the application.

There is also a folder name **common** which has utility classes and functions that are implemented to be used by all the entities within the application.

3. OOP principles

Tried to follow most of the implementations done with OOP principles. **Views**, **Serializers**, **Managers**(reusable classes to perform DB queries) and **Helpers** (reusable classes to perform generic algorithms, calculations etc)

4. Scope to extend classes for future

Common classes in helpers like **payment, email, and ballot sector algorithms** are implemented in a way so that in future they can be extended. Also, new sibling classes can be introduced by using **abstract classes** which can replace other sibling concrete classes from client code without breaking any feature.

5. Custom authentication for custom api token

A very simple and customized API authenticator is implemented to check and authorize user API requests by providing **api_token** in the request header.

6. Unit test

Unit testing is a very important aspect to reduce bugs and identify feature breaking for any particular changes. For showcase purposes, some parts of the codebase are covered with unit tests.

7. Meaningful and customized logs and exception handling

Set up meaningful and customized logging mechanisms and exception handling in settings. By this, finding the root cause of any type of exception becomes much easier and faster to solve.

8. Serializers

Both default and customized serializers are used. Serializers are mostly used to serialize, deserialize, parse and validate data that comes within the request body. The good part is that all the class methods can be overridden to customize as the developer's choice. For example, in some areas validate method has been overridden to do custom validations.

Things omitted but prefer to use in real life

1. Used Django default SQLite database. In real life, I would like to use Postgresql as Django documentation and the community prefers this database more saying Postgresql is much more supportive of the Django framework than other databases.

2. Would like to use caching mechanism in case more optimization needs for scaling purposes in future. My personal choice for the cache is always Redis as it is extremely faster, can be also used as a persistent datastore and comes with varieties of data types and data structures.