Intro

Start time: 13:35 19.3 CET

Take home task has been given by a company Access Members. The task requires me to create an event ticketing app featuring **event lists, purchase tickets, orders**.

The time frame for the given task is 2 days. 48 hours.

The first approach will be to read through the task thoroughly multiple times not to miss anything after which I will list the stack used. Create tasks which will be done. Each task will have a problem and proposed solution which will be iterated over multiple times. I will decide how long each step (task) lasts to focus only on the most important parts and get the app going.

Stack

Backend: NodeJs w NestJS, Postgres w Typeorm. GraphQL.
Frontend: React Native, Redux, NativeBase

Tasks (Monolithic app)

- Task 1: Design the database schema, **ETA** 2.5 hours
  - Start with requirements and brainstorm forward
  - Use https://dbdiagram.io/ so that the schema is easily visible by another person
  - Reiterate and see potential problems that could arise
  - See where optimizations can be applied
- Task 2: Bootstrapping monorepo, **ETA** 1.5 hour
  - Init github repo
  - Init new nests project in backend folder w required dependencies
  - Init new react native project in frontend w required dependencies
  - Add docker compose for easier development
  - Add code styling rules
  - Write a clean and beautiful readme file
- Task 3: Building backend **ETA** 8-12 hours
  - Db setup, entities
  - Apollo set up
  - Build everything around users
  - Build everything tied to listing events
  - Build everything tied to purchasing tickets
  - Build order confirmation
  - Test everything and proceed and proceed to frontend

Task 2 Frontend (later)

Db Schema

Overview:
There must be a **User** which is pretty much always the core of every app. Then the app is based around something which in this case is **Events**. Events always have **Tickets**. Events **always** have tickets which must cost money in this context otherwise this app can't make money and has no reason to exist. (Not an advertising app for events). An app to **purchase** tickets. Upon **purchase** an **order** is created displaying information. Thus, main entities are defined as follows:

- **User**
- **Event**
- **Order**

A user can have basic info like Name, Email and Password.
An event must have seats and seats taken. Arguably you could start with X amount of seats and decrement to zero but this removes the information of how many people are actually going so 2 columns is okay. Additional info is of course Name of the event, date an entity is created and data of the actual event.

Users can purchase N amount of tickets on a given event as long as there are available seats. Users can purchase tickets to multiple events. This is visible in the **orders**. However as the task states the solution should be designed to be future proof. Thus an **order** is simply a record of a transaction, a ledger so to say. These transactions can be refunded, cancelled or be in a failed state. Also tickets can be moved across users, say **gifted** to a friend. So it's safe to say that you don't gift an **order** but a ticket itself. So as stated once the order is completed the user is given **N** amount of tickets. These tickets can be gifted to a friend, gifted by an event organizer, and can be revoked. So now the entities are **User, Event, Order, Ticket**. The task states that **order** should have column **event details**. I am against that. For starters I would have a **fk** that links events and orders.

Features and problems?
- What if I want to refund 5 out of 20 tickets? How will that affect order?
- What if an organizer gives tickets? Order is not created?
- What if there are VIP, Gold, Silver tickets? Can a user buy a mix of those? Unlikely but possible.
- What if I give a ticket to another user? What if I give multiple tickets to another user?
- Are tickets destroyed on refund? How will this affect event and order records?
- What if a buying user wants to refund gifted tickets? System design, but still how will it affect the db? Is the ticket destroyed or kept in a refunded state? On which user is the

refund created? (System can easily support this so it's less important). A ticket would be nullified and order of type refund created.
- Prevent refunding gifted tickets by the organizer? (easy, ticket will have null fk on order)
- Can a ticket be gifted to a non-registered user? How will this affect the record in a db? Do we start with nullable or non nullable fk on user?
- Do we refund based on the price bought or on current price? (increase due to event approaching) - Not really a db issue (but will keep it)
- Can the system be expanded to use promo codes, discounts, loyalty programs?.
    - Promo codes can be a separate entity and applied on order. Promo code can track which order it is a part of. Could a loyalty program work? Orders would still apply? No?
- Can the system be expanded so that a single order can buy tickets for multiple events?.
    - It would require and extra table that has order and event_order which would track multiple events across a single order. A ticket would then need to have "event" and "order" then. It can work for gifted and bought tickets. (order nullable). Can the data be migrated from this system to a system with event_order? Yes.
        1. Create event_order.
        2. Populate event_order with data from the order
        3. Remove the fk column on order which links it to event

Ok, lots of brainstorming so here is a proposed solution.
User has an id, name, email, and pass
Event has id, Name, Description, Ticket limit, Tickets sold.
Order has id, event id fk, cost, number of tickets (for easier queries)
Ticket has id, user id fk, order id fk, event id. (order has events so not needed).

Event needs a pessimistic lock on tickets sold. It's okay if upon reading you don't see correct information, this is acceptable. However, overbooking isn't so update lock is needed. Users seeing the info will get him to buy the ticket and here is where locking is needed. Pessimistic to avoid retries.

Ticket can have massive writes if the event is hot. But since a single user buys multiple tickets a copy command can be used in postgres to speed things up. It is expected to handle small events just as well as large concerts. Use the archive once the event expires. Partitions could be complex due to the difference in size of events so leave that for later.

Order and **User** don't need special attention. I would only add index on order by user.
Ticket_User id, user_id (index), ticket_id. (denormalized for user reads).

Tickets added through transaction. (Ensures both Ticket and Ticket_User are updated).
And now read is super fast because of index and insert is super fast as it has no index and uses copy.

This concludes db setup. This should allow lots of writes and reads where necessary and prevent overbooking with locking mechanisms. It is also possible to add new features.
Due to time constraints I will build the schema now in an online app.

Schema: Ticketing system db schema

Analize.

Users can view events. (List items from event table).
Users can purchase tickets for a single event.

- Pick event, select ticket count
- Purchase -> order created, tickets added to db
- See the tickets and order from db. Events, Order, Ticket_User table

Users can see which events are sold out and how many tickets are left and how many people are going. Event.total_tickets and Event.available tickets.
Order confirmation. Order added, tickets added -> complete transaction and show it back to the user.
All requirements are met! **View events, purchase tickets, sold out events!, order confirm**

Task 1.1 completed!

Time 16:46 - 13:35 -> 3 hours 2 minutes, 40 minutes above estimation.

Rest Starting 16:46
Rest end 19:03

Starting task 1.1 Bootstrapping
Created repo on github private with no files
Ran git init in root, set origin, set upstream
Installed and ran nestjs, installed graphql apollo and ran first query
Failed to bootstrap the app. Restarting.
Ran nest new backend. Large number of issues.
But what got me was the fact that `migrations: ["src/migrations/**/*.js"]`
Was targeting .ts files instead of .js :(. Any way. Waaay beyond time limit:

End: 23:34 -> 4.5 hours

Task 1.3 Building backend Starting 11:55 Thursday 20 march

// Shortcut chatGpt prompt

nest command to generate new modules i need to generate modules with just entity and module file

—----

ok i have ticket order event generate commands for all that

1. Created plain entities
2. Basic create and get users

Mini pause (cooking) 13:28
Pause ended 14:19

3. Create update list events
4. Buying tickets flow/order flow, prevent overbooking, lock mechanisms and tested manually

Small pause 16:05
Pause ended 16:12
Small review.
Users can order tickets, users can list events. Now what is left is to see the tickets by user.

5. Fix order flow (single ticket on n order, order not added to ticket)

This appears to cover all features. Even though the backend is still bad. No proper logging, no custom clean explicit errors, no auth, pass hashing, tests, env vars. Frontend must be started. So in a sane amount of time at least the MVP will be finished.

Now the last thing that I wasted most of the time was bootstrapping the backend. I will check for the frontend if there is something that matches the stack needed and simply clone the repo. Maybe there is an app similar to this. Either way, I will try to bootstrap as quickly as possible.

Pause: 16:34
Pause end 16:50

https://github.com/zerodays/react-native-template
This one is the pick, it has state management (zustand, haven't used but could switch to redux easily). Tanstack query, nativewind… Should pretty much handle everything.

Can't install infiscal. Will try plain expo
Yep, expo is pretty good. App running

Haven’t reported in a while but going strong.
List events, Event detail, Pull to refresh, Navigation. Will take a small break.
19:30 pause
20:00 unpause

Massive changes, with navigation, header, reload, pull-to-refresh, migrations, creating events, listing events, details. Etc:

Next thing is buying tickets, Readme, docker-compose support. Which would conclude the app. Rn, i will check how much total time have i spent.

15h 28m total time to now, which leaves me with 32 minutes to configure ticket buying and README. After which I will push the commit, which is of course visible in the repo. That would give me 2 work days. Which is the minimum time required, if successful I have completed the app at least the listed features.

So what’s left is. Show sold out (in a better way).
Order and README.

Also commits should show my work time so I guess there is no need to worry about when I did what right?

21:25 pause
21:34 f it, unpauseee

Doneee!!!! Readme left :)))