SIMVEST Backend – Planning Log

Core Concept Recap

- Each team represents a company.
- Teams can **buy shares** of other teams (aka companies) using the money they're allocated.
- Every team starts with some base valuation (say 100K) and 1000 shares at 100/share.
- Buying shares from a company increases that company's **valuation**.
- The frontend will show valuation data and share distribution live.
- All buying/selling actions happen through the ~~discord bot~~ **React frontend**, with requests going to the backend (PostgreSQL mostly).

Auth

- Each team gets a **unique team ID** (or login code) to access the frontend(as discussed in the meet).
- Auth system will be simple: either **OTP-based or pre-registered logins.**
- On login, they get access to the **buy interface + portfolio view.**

We'll probably go with **token-based login** for ease (JWT or even session cookies).

Database (PostgreSQL)

PostgreSQL makes sense here for a couple of reasons:

- We need **row-level locking** for **real-time concurrent** share buys.
- Structured schema is easier to work with for transactions + aggregations.
- Great support for concurrency, `FOR UPDATE`, and ACID compliance.

Tentative tables:

`teams`

- `id`
- `name`

- `valuation`
- `shares_remaining`
- `initial_money`
- `has_taken_loan` //(no idea if there's any loan system, this was GPT generated)
- `loan_amount` //(same)
- `last_transaction_time`

`transactions`

- `id`
- `buyer_team_id`
- `seller_team_id`
- `num_shares`
- `actual_shares_received`
- `dice_result`
- `valuation_change`
- `timestamp`

`portfolios`

- `id`
- `team_id`
- `invested_in_team_id`
- `shares_owned`

Share Buy Logic

When a team initiates a **buy request**:

1. Buyer selects how many shares they want to buy from a particular team.
2. Frontend sends POST request to `/buy` (for e.g.)
3. Backend does the following:
   - Locks the **seller's row** with `SELECT ... FOR UPDATE`.
   - Checks if seller has enough shares remaining.
   - Deducts money from buyer's funds.
   - Updates seller's shares.
   - Updates buyer's portfolio.
   - Adjusts valuations of both teams using the **valuation formula** (below).
4. Commits the transaction. Unlocks rows.

This guarantees that **no two buyers can simultaneously buy the same share stock** from a seller one will wait for the other to finish. (still some issues with this)

Dice Roll Outcomes

ABSOLUTELY NO FUCKING IDEA WHAT TS IS

Valuation Update Logic

One basic formula which we'll be using:

$$N = B \times (1 + C \times 0.3) \times (1 + S \times 0.002)$$

Where:

- `N` = new valuation
- `B` = base valuation
- `C` = number of companies that invested in this team
- `S` = total shares bought from this team

Concurrency & Collision Handling

This is probably one of the **core technical challenges** in the backend.

The Problem ??????????

Let’s say **multiple teams try to buy shares from the same company at the exact same time** — like Team A and Team B both trying to buy from Team X, right down to the same second.

If we’re not careful, this can lead to:

- Overselling
- Incorrect share deduction
- Messed up valuation updates
- General chaos in the database
- And a badly fucked up situation on the D-day.

This is basically a **race condition**, and it's really easy to mess up when multiple users are sending simultaneous requests to the backend.

The Current Plan :D

We're using PostgreSQL's native support for **row-level locking** to handle this.

Here's how it works:

1. When a team sends a request to buy shares from another team (say, Team A buys from Team X), the backend starts a **transaction**.
2. It runs a query like `SELECT * FROM teams WHERE team_id = X FOR UPDATE` — this **locks** Team X's row in the database.
3. While this lock is active:
   - No other transaction can read or write to Team X's row until this one finishes.
   - So if Team B also tries to buy from Team X at the same time, **that transaction will wait**.
4. Once Team A's buy transaction finishes (after applying the dice logic and updating shares/valuation), the lock is released.
5. Only then will Team B's transaction proceed, now working with the updated, accurate data.

Still Needs Testing Though

This seems solid in theory and PostgreSQL is pretty good at handling this kind of concurrency but we **still need to test it out** during a dry run with:

- Multiple buyers
- Simultaneous clicks
- Randomized dice rolls !?!?!?!?!?!?

We'll need to see if:

- There's any noticeable delay or blocking
- The locks cause any bottlenecks with many teams
- We ever run into deadlocks (hopefully not, but good to watch for)

If this turns out to be too slow under load, we might consider queueing requests, but for now, the row-locking strategy looks clean and manageable.

React Frontend

It should show:

- Live valuations (sorted leaderboard)
- Each team's share count
- Buy interface (dropdown → pick team → input shares → confirm)
- Portfolio view (who you've invested in, and how much)

It'll refresh valuation data every 10–15 seconds (can use polling or WebSocket later).

HMMM???

- Do we add a feature to **allow teams to view other portfolios**?
- Do we show valuation breakdown (like pie chart: how much came from which teams)?
- Need to test if **PostgreSQL scaling holds up** if many teams click buy at once!?
- And what is that **dice roll shit**!?(have to catch up with the events domain)

Remaining Work

- [ ] Implement **buy shares** logic with PostgreSQL row-level locking (`SELECT ... FOR UPDATE`)
- [ ] Finalize and apply **valuation update formula** after dice roll
- [x] ~~Set up **team login/auth** using unique IDs~~
- [ ] Design and build essential **API endpoints** (`/buyShares` , `/getPortfolio` , `/getLeaderboard`)
- [ ] Enable **real-time data updates** (polling or WebSockets – to be decided)
- [ ] Add backend **validations** (one buy at a time, limits, loan rules, etc.)
- [ ] Perform **concurrency testing** with simultaneous buy requests