CISC 320, Section 4: Project Documentation

Project name: Ticket Trading

Team name: Team M

Git repository link: https://github.com/marco-dm1/CS320-Team-M
Final deliverables video:
https://drive.google.com/file/d/1ou_xIZTg72MijrjaHomDALr1kP_KVly0/view?usp=drive_link

Git usernames:

- **Victor @Victor-Jedamanov**
- **Kai Zheng @iJellow101**
- **Himnish @himccoder**
- **Ron @Ronkleinhause**
- **Aareb @akchdhry**
- **Marco @marco-dm1**
- **Ahmed @ahmedar04**

1. Requirements

1.1. Overview

- A web application that allows individuals/corporations to purchase/resell tickets.
- First a user is asked to give their details for creating an account and for safety of the app/transactions.
- They are taken to a page which shows their personal details and balance money they have in their account.(They can choose to add money(We will deal with just numbers at the moment, not actually connecting to real bank account as of now))
- A buyer places a bid(limit order-{event, bid price, qty of tickets}). This bid is placed into an orderbook.
- A seller can place an offer of a quantity of tickets at a certain price. This also gets placed in the orderbook. (Orderbook is a table that is a list of current bids and offers, the user can see it and can accordingly decide what to bid and what offer to place or to change a bid/offer).
- Now all bids and offers for the specific event are compared and when a price is acceptable to the buyer and seller. The transaction takes place. Before transaction, buyer is asked to confirm proceeding with the transaction. The money is deducted from buyer's account and added into seller account.
- Ticket object{}(seller) → Ticket object{}(buyer), the ticket transfers from seller to buyer and then is deleted from seller's account.

- For e.g. if offer is cheaper than what buyer was willing to pay(bid) and there is enough quantity, then transaction occurs. Otherwise, if offer was more expensive then the buyer can decide if he/she wants to be willing to pay more or decrease quantity.

1.2. Stakeholders

- Ticket buyers.
- Ticket sellers.
- Venue/Event holders
- Development team

1.3. Features // At least 6

- Create user accounts
- Log in/ Log out
- Letting users to auction tickets
- Letting users to bid on tickets
- Letting event organizers/venues adding events to the system
- In app “wallet”
- Exchanging the tickets between accounts.

1.4. Functional requirements (as user stories or use cases) // At least 6

- User account creation:
    - Description: Users can create an account
    - Actors: Ticket sellers, Ticket buyers
    - Trigger: “Create Account” button on menu screen
    - Precondition: User is on menu login screen
    - Success End Condition: The user does not have an account and successfully creates an
    - account with a given email and password.
    - Failed End Condition: The user cannot create an account and an appropriate error
    - message is displayed. Failure can occur because of… invalid email address, invalid password, or email has an account already associated with it. If the email address is invalid, an error message will be displayed notifying the user and prompt for a different email. If the password is invalid, an error message will be displayed with the valid password conditions (alphanumeric characters, at least 1 capital letter) and prompt for a different password. If the email has an account already associated with it, redirect to a login page.

- User login or Event/Venue login:
    - **Initially,** the entity must access the web application in their preferred web browser. Once the individual or Event/Venue successfully accesses the landing page, assuming that the user chooses the option for “account login”, they are then redirected to a separate web page where the user is prompted to enter in their **username** and **password**. And so now,

for the **success end case**, if the user enters in the correct credentials that match with the one-way hashing/encryption, works in a way in of which the password entered will be hashed in accordance to a hashing algorithm of our choosing and if it successfully matches with the hash provided in our system, then… (**WIP:** If the user enters in the correct username and password as cross referenced with our database, then the user will be successfully logged in and will be taken to a separate web page with the Homepage). However, if we're to consider the **failed end case**, if the user fails to either provide a valid **username** or a valid **password** then the web application window remains on the same page and presents an error message in accordance with the information that was incorrectly entered. The error message will entail either an error message expressing that the user has entered an incorrect username or password.

- User Sign out or Event/Venue Sign out:
    - The location of the Sign Out option for the users is still to be determined; however, we envision that the overall process will be essentially located on the same web page as when the user first logged in. Essentially what will happen is that when the user clicks on the logout button located in **TBD**, they'll be presented with two different options that essentially asks the user if they're certain that they want to be "Sign Out" or "Cancel". If the user continues to proceed with the "logout" option, then they'll be taken back to the landing page where the user will be prompted with a message ensuring that they've been successfully logged out. The general message will be along the guidelines of "You've been successfully signed out of your account.". Now if the other case were to take place where the user perhaps accidentally pressed on the "Sign Out" option, then they're also able to cancel this process and not proceed with the signing out of their account. If the "Cancel" button takes place then the user's web application window remains on the current page or in this case the Homepage.

- Portion of the Homepage: **Buy Ticket and Sell Ticket**
    - Description: Users of the platform can choose whether or not they'd like to buy or sell tickets
    - Actors: Users, Venues, Event Holders
    - Trigger: User clicks on either one of the options and proceeds to select which option/ticket price they'd be willing to buy/sell from
    - Preconditions: User is logged into an existing account and has verifiable ticket information that is then cross-referenced with the ticket purveyor for authenticity (unsure of how)
    - Success End Condition: The *Actors* successfully list their ticket price in of which they're comfortable at selling at and the ticket is now being circulated amongst other users or in the scenario in of which the user has successfully purchased a ticket, the user will then be able to access the ticket (process as to how or where is still TBD) and able to utilize the ticket since it has now been placed under their name (perhaps use a time tracker system in order to avoid seller duplication and also provide an option for buyer protection)
    - Fail End Condition: The *Actor* who wants to sell a ticket fails to provide a valid ticket confirmation number which has been cross referenced with the third party provider, and

is thus rejected from being able to post this particular ticket in the marketplace. Or perhaps the individual is attempting to purchase a ticket at a certain price point; however, fails to do so because of insufficient funds in the user's account

- Steps:
  User selects one of the options (e.g. Buy or Sell)
  - If Buy is selected. Then the user is prompted to select which ticket (potentially using a cart option where the user can add objects into cart) they'd like to purchase. If the user's payment method of choice processes then the ticket is successfully added to their account
  - If Sell is selected. Then the user is prompted to enter into the fields a (TBD) a immediate-sell-price, bidding opt-in, and a valid ticket confirmation number (this is cross checked/referenced from third parties)

- Portion of the Homepage: **Stock Listings for Sell-Tickets or Bidding-Tickets or Both**
  - Description: Shows the listings of the price of every ticket currently in circulation in addition to showing the bidding for tickets currently being bidded upon
  - Actors: User Sellers, Venue Selling, Event Holders Selling
  - Trigger: Individual purchases a particular ticket from the sell-ticket area or the person makes a particular bid in order to achieve receiving the ticket at a lower price.
  - Preconditions: Checks whether the current ticket being sold is in stock, and also if the currently being bid on ticket is still available. And as per usual that the user trying to make a purchase or a bid has to be logged into an existing account
  - Success End Condition: If an individual wants to purchase a particular item, the item has to first be checked if still currently available, afterwards, a purchase window screen will appear, and once the individual's payment method has been successfully authorized, then the ticket is then removed from the circulation and will be placed in the wallet of the buyer. In the case of bidding, the individual must initially enter a price higher than the current price, and if this price point has been reached, then the bid is then processed and a temporary hold of the amount bid is placed on the account until either the time expires or another user bids an amount greater than that of which was previously bid. If the ticket's bid has been successfully reached at the point of which the seller had previously indicated they were comfortable with selling, then the purchase gets processed, and the ticket gets removed from circulation and gets sent to the buyer's ticketing wallet (possible implementation)
  - Failure End Condition: If an individual doesn't have the funds to either bid or make a purchase, then the entire operation fails and an error message will appear informing the user that "There's been an error processing your payment, please try again later". In addition to that, if there's been perhaps some error where the ticket is still in circulation despite being out of stock or having been already been purchased, then there'll be an internal server error, and an error message will pop up for that too informing the user that the ticket that they're attempting to purchase has been removed from circulation.
  - Steps:
    - If a user attempts to purchase from the SELL category, then they'll be brought to a screen for them to enter their preferred form of payment, and once the purchase

goes through, then the user will receive an order confirmation in addition to the actual ticket appearing in the wallet.
        - If a user attempts to make a bid from the BIDDING category, then they'll be prompted to make an appropriate bid that is higher than that of which the price of the bid is currently at, and if the price is higher, than the price is accepted and the bidding price is updated and a temporary hold amount is poised in the account of the bidder. Once the price of the bidding ends/expires or the bidding is overtaken then the poised hold is removed or is finalized and if finalized then the individual's price is then posted onto the account

- Portion of the Homepage: **Search functionality**
    - Description: Users can search for the type (location of venue, artist, etc.) of ticket that one is hoping to purchase
    - Actors: Users, Venues, Event Holders
    - Trigger: User inputs something in the search box and searches
    - Preconditions: User should be logged in to access the search function box
    - Success End Condition: The list of relevant events/artists performing should be displayed in a row-column based on the total number of relevant results in accordance to the search.
    - Failed End Condition: No list appears, and a default search message will be displayed informing the user that there are no search results that are similar to that of which the individual has searched for. An error message along the guidelines of "Sorry, but there are no events/artists of your searchings".
    - Steps:
        - User searches in the search box in location (TBD) and then presses search or enter whichever the user prefers. Then, if certain options are available based on the keyword search, then the user is presented with these options; however, if no options seem to persist, then the user is returned with an alternative/default message of no search results are available based on the searchings of the user.
    -

1.5. Non-functional requirements // At least 3

- Ticket Value Approximation:
    - We could add a feature that attempts to approximate the market value of a ticket so that users can compare it to what the seller is selling it for. This can help users see if the prices are hiked up unreasonably.
- Real Time Offers/Bids:
    - We could add a feature that displays all the ticket offers and bids for a desired event and show changes in real time. This would allow users to look at a large number of relevant tickets quickly while also being able to take other peoples' bids into consideration for their own selling/buying.
- Maintainability:
    - By structuring and labeling the code in a well-documented, easily understandable way, we'd be making it much easier to maintain in the future. If this application were to have

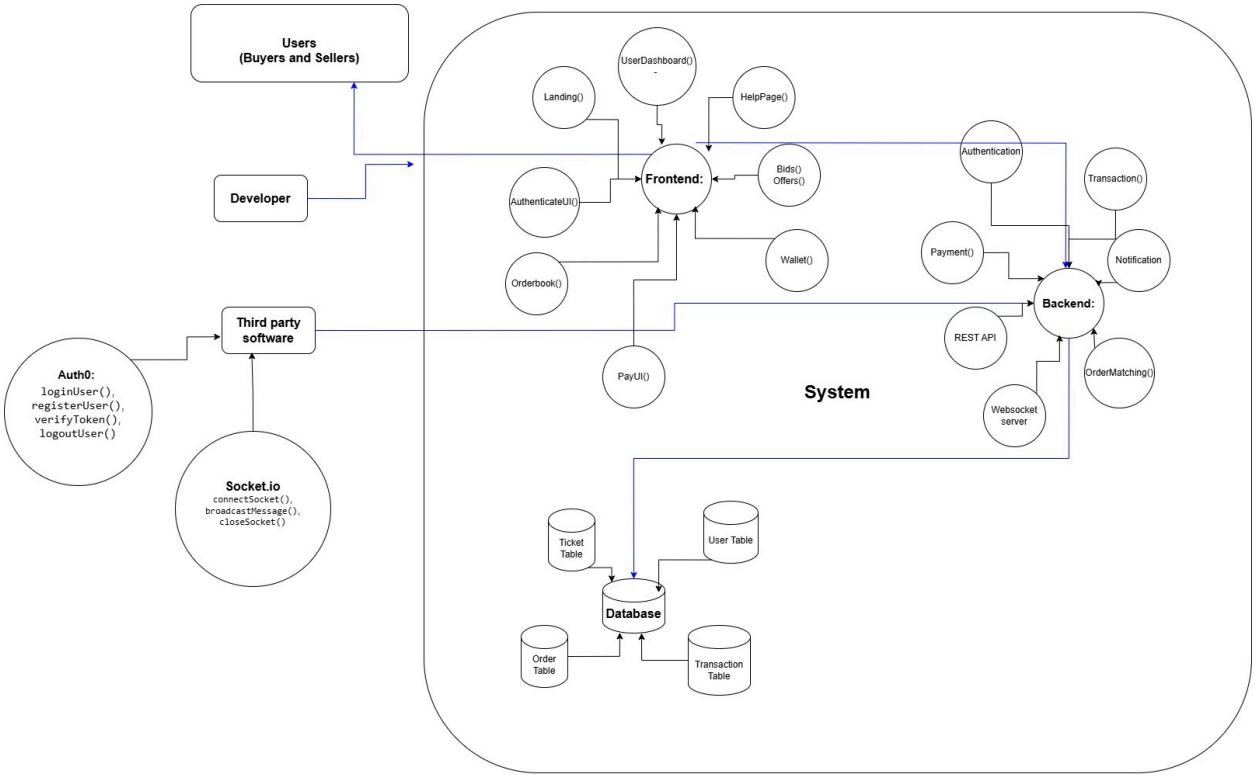
extended use or need bug fixes, ensuring that the structure of the project is easily understandable would greatly decrease time spent on any necessary future changes.

1.6. Challenges and risks // At least 3

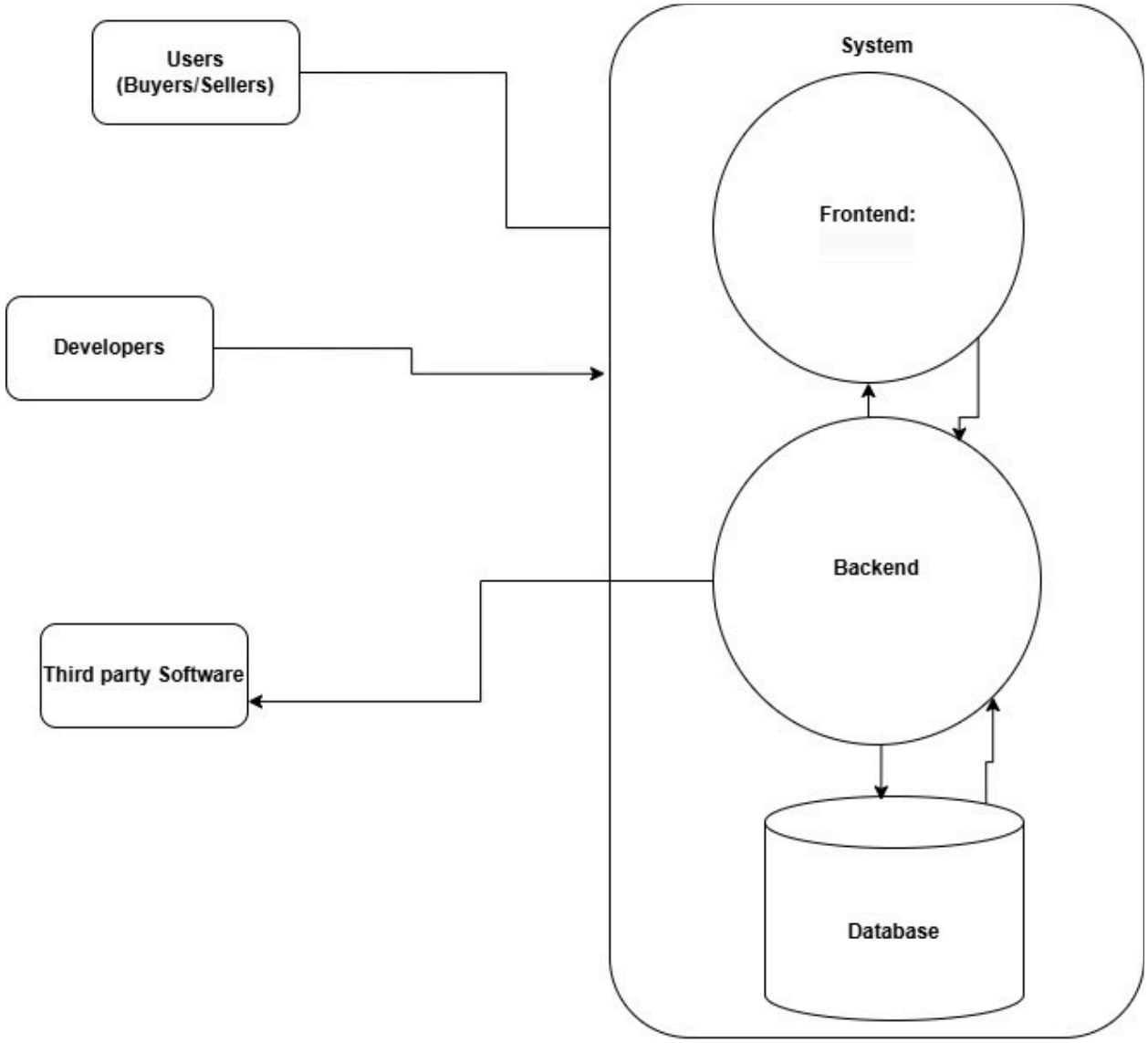
- There are huge legality concerns. Venues might have policies against reselling and void tickets, and regions might have laws involving price hikes from reselling. We'd have to check with venues and local laws wherever the application is used to make sure everything is legal.
- Ticket authenticity is another concern. Users might sell tickets to fake concerts, fake tickets to real concerts, or tickets that might require an ID check with the original buyer. We'd have to implement a method to ensure the tickets sold are actually usable and able to be transferred to the buyer.
- Ticket transfers might be an issue as well, since the seller could be paid the desired amount and then simply refuse to transfer the ticket. If the tickets are physical they'd have to meet in person to exchange them, and if the tickets are digital then we'd be relying on a different application to make said transfer.
- We'd have privacy and security concerns since we'd have user logins and a database. If this information got breached, their logins would be compromised and sellers could be made to sell digital tickets for free. Any information stored in the database would also be leaked.
- Allowing multiple users to use app real-time, concurrently.

2. Design

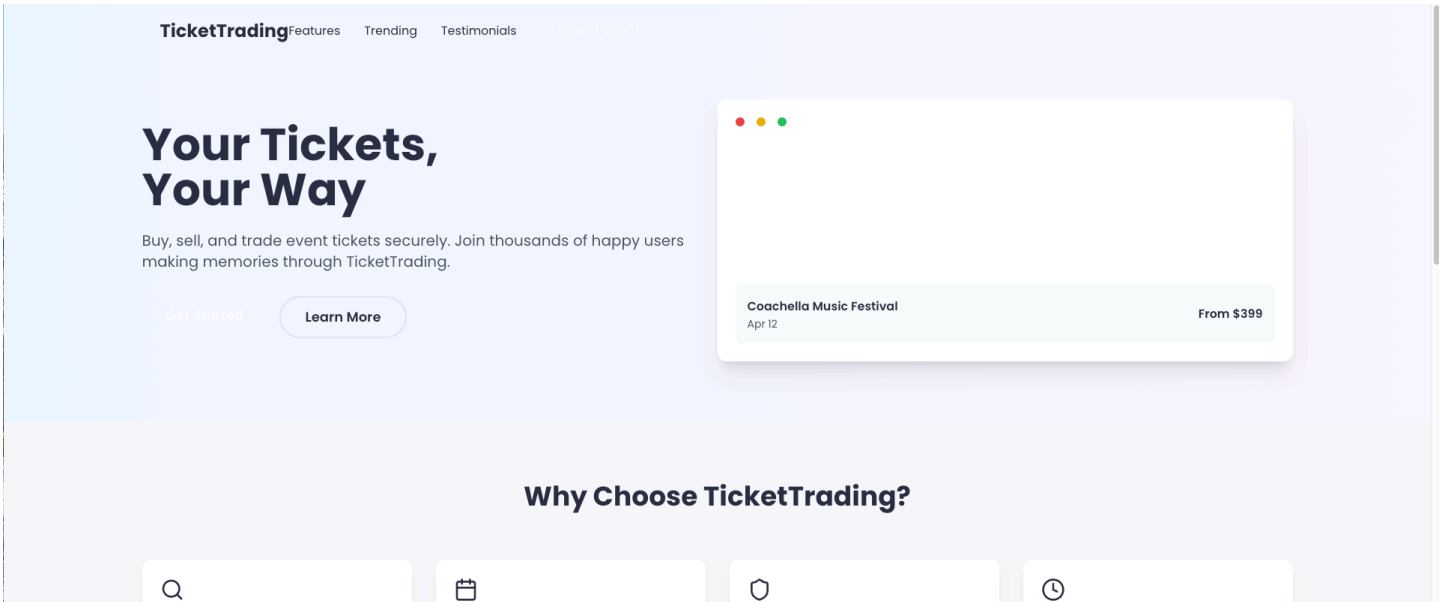
2.1. Architecture

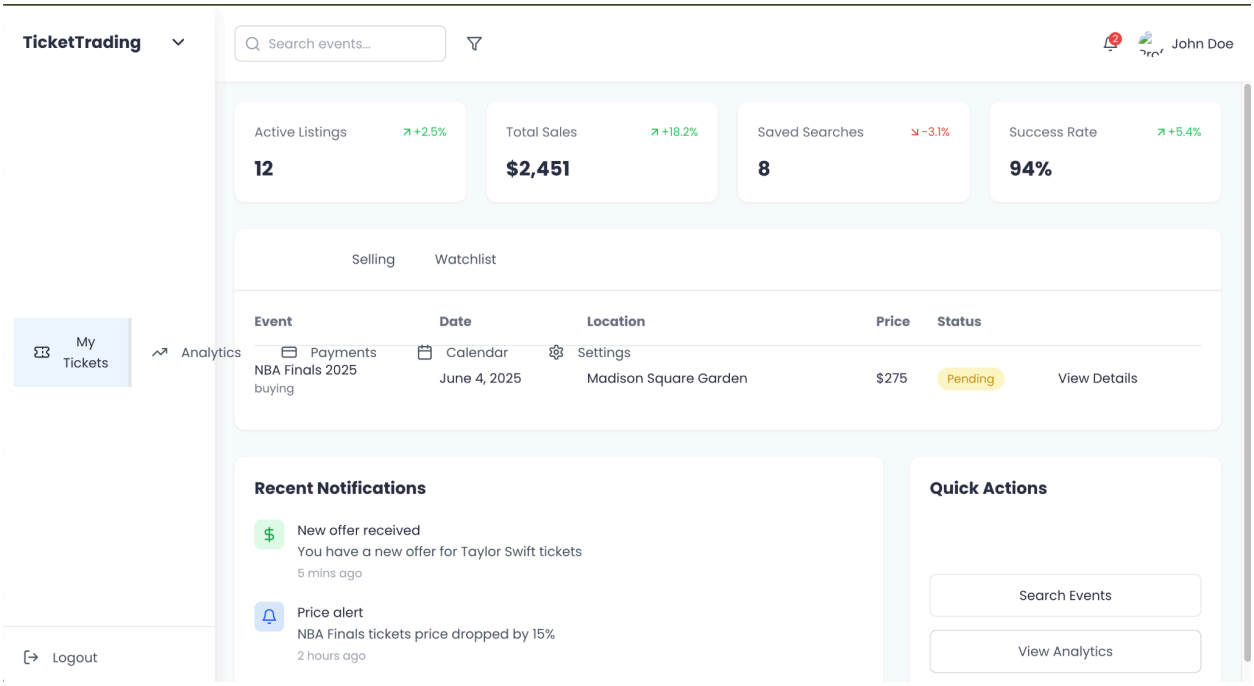


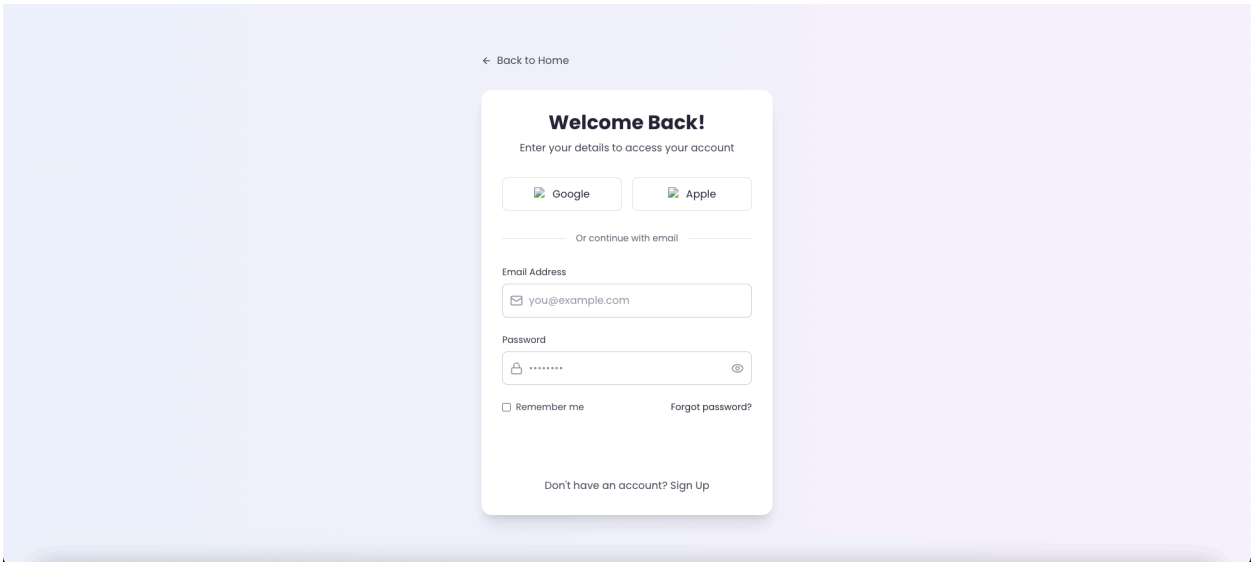
2.2. Low-level design

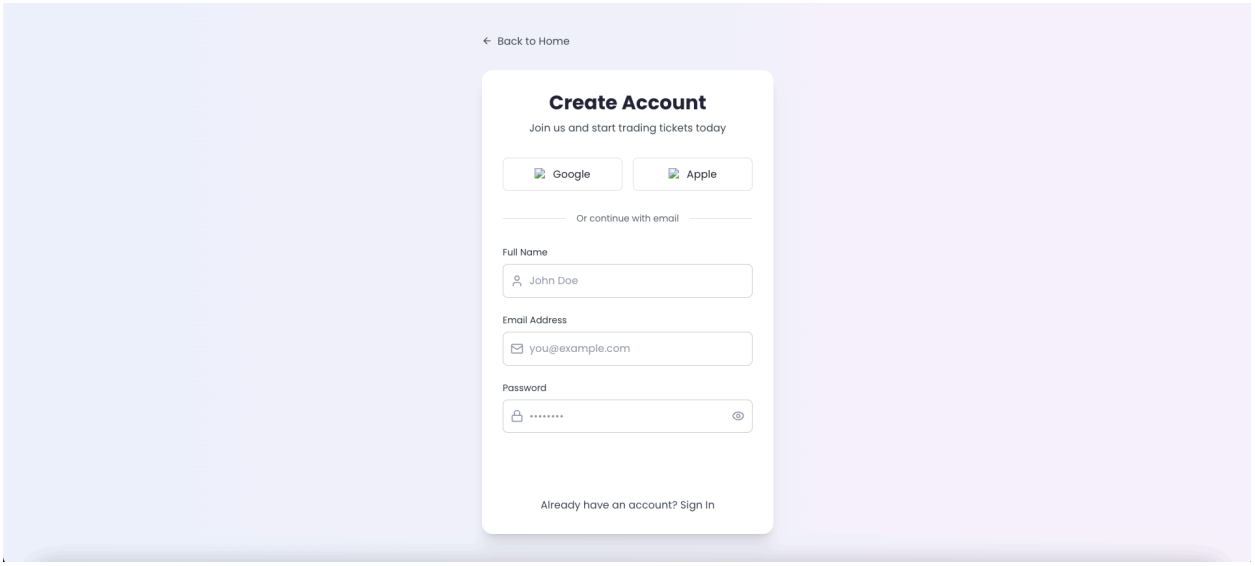


2.2.1. UI Design




TicketTrading
Search events...
John Doe
Active Listings
+2.5%
12
Total Sales
+18.2%
$2,451
Saved Searches
-3.1%
8
Success Rate
+5.4%
94%
Selling
Watchlist
My Tickets
Analytics
Payments
Calendar
Settings
Event
Date
Location
Price
Status
NBA Finals 2025
buying
June 4, 2025
Madison Square Garden
$275
Pending
View Details
Recent Notifications
New offer received
You have a new offer for Taylor Swift tickets
5 mins ago
Price alert
NBA Finals tickets price dropped by 15%
2 hours ago
Quick Actions
Search Events
View Analytics
Logout





2.2.2. Application Programming Interface (API)

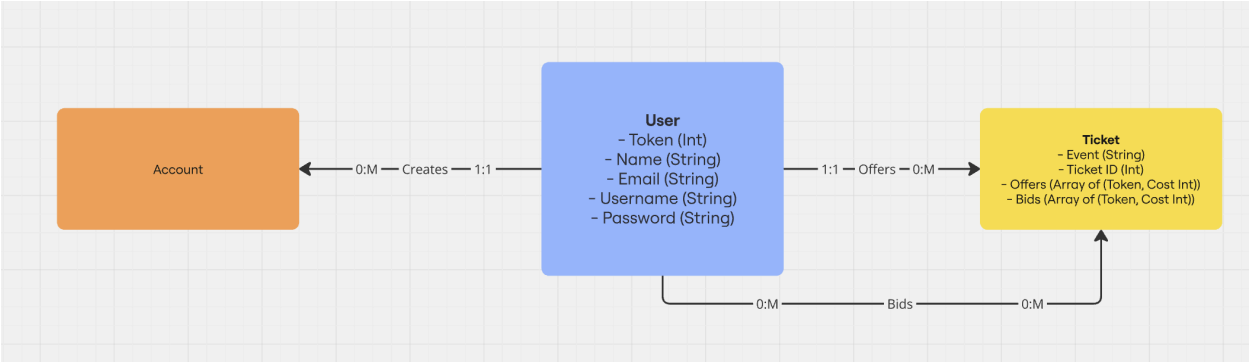
1. User Account
    a. Create account - /api/user-account/create-account
        i. Input: Username, password, email
        ii. Output: Returns True/False. If False then state the reason for failure. Reasons may include: empty fields, email already in use, username already in use, weak password, timed out error
    b. Login - /api/user-account/log-in
        i. Input: Username, password
        ii. Output: Returns True/False. If false then state the reason for failure. Reasons may include: empty fields, username or password are incorrect, timed out error
    c. Log out - /api/user-account/log-out
        i. Input: Authentication token

ii. Output: Returns True/False. If False then state the reason for failure. Reasons may include: unknown log out error

d. Get account data - /api/user-account/get-account-data
   i. Input: Authentication token
   ii. Output: Returns account data in JSON format on success. Returns “error in retrieving account data” on failure.

2. Ticket Data
   a. Post ticket offer - /api/ticket-data/post-ticket-offer
      i. Input: Authentication token
      ii. Output: Returns Ticket ID on success. Returns “error in posting offer” on failure.
   b. Post ticket bid - /api/ticket-data/post-ticket-bid
      i. Input: Authentication token
      ii. Output: Returns True on success. Returns “error in posting bid” on failure.
   c. Get ticket offer - /api/ticket-data/get-ticket-offer
      i. Input: Ticket ID
      ii. Output: Returns ticket offer on success. Returns “error in retrieving offer” on failure.
   d. Get ticket bid - /api/ticket-data/get-ticket-bid
      i. Input: Ticket ID
      ii. Output: Returns ticket bid on success. Returns “error in retrieving bid” on failure.

2.2.3. Data model (e.g., class diagram, ER diagram)



2.3. Tech stack with justification

1. React.js
   a. React is a JavaScript library developed by Facebook for building modern, dynamic user interfaces (UI). It uses a component-based architecture to make building complex front-end applications more manageable. React allows you to build responsive interactive UIs efficiently. It utilizes a virtual DOM, essentially meaning that it renders parts of the UI that the user changes. React’s component-based architecture encourages code reusability, which speeds up development time. React’s ecosystem which includes powerful tools like React Router and Redux which enables us to create feature-rich single-pach applications with ease. Is extremely common in the industry, so there’s more

easily accessible documentation in case we run into issues. Also, javascript is a language a majority of our team is comfortable with.

2. Node.js
    a. Node is a JavaScript runtime that allows you to run JavaScript code on the server side. Node enables very agile execution of JavaScript outside the browser. Node is known for its excellent choice building for real-time applications, handling repetitive requests efficiently, and also delivers high performance. It's also a very good option primarily for JavaScript developers, which as one can see from the stack, that it's very much so beneficial for us because the server and client side both use JS. And so, by using Node, one can build and deploy fast/scalable server-side applications that can handle thousands of concurrent connections without hindering performance. We were deciding between Node.js or Django and decided that having both front end and back end using javascript would be easier for our team to handle.
3. Express.js
    a. Express is a minimalistic, flexible Node.js framework that helps create robust APIs and web applications. It provides a robust set of features for building single-page, multi-page, and hybrid web applications. Express simplifies the process of handling HTTP requests, routing, middleware integration, and etc. Express allows us to build RESTful APIs and web applications fairly quickly by providing essential functionalities like routing, middleware, and session management, thus in return helps us developers focus on building more features rather than the boilerplate code. It's also very lightweight, fast and highly customizable, hence making it ideal for building scalable back-end features. This works well with individuals in our group who already have a good amount of familiarity with Node.js, and this basically adds an additional layer of optimization.
4. MongoDB
    a. MongoDB is a NoSQL, document-oriented database that stores data in flexible, JSON-like documents (BSON). MongoDB is especially useful for applications that require flexible schemas or working with large volumes of unstructured data. MongoDB allows one to not have to predefine tables and schemas, this allows us to essentially adapt and change our data structure as our application grows. MongoDB's ability to store data in a format similar to JSON, which works synchronously with JavaScript (used in both the front-end and back-end), further simplifying data manipulation. Most of our team doesn't have experience with SQL, leaving us with MongoDB which uses Python. Also, Python is a language the majority of our team is comfortable with.

2.4. Challenges and risks // At least 3

- Real time update during transactions
    - Moving tickets directly between users at a moment of transaction can be complicated, taking in consideration that we can have multiple requests for the same bid/offer at the same moment and that a ticket cannot be sold to more than one person. We will need to implement real time transactions, that will prevent any mistakes.
- Authentication
    - Making sure that our users are real humans and that no other identity can gain access to the account of the user is highly important, especially when money is involved, this

process can be challenging. We will need to authenticate our user every time they log in to their account.

- Token handling(security)
    - In order to move tickets between users securely and efficiently we aim to use tokens to identify each unique ticket, working with that system can be challenging.
-

2.5 Customer Feedback

- Perhaps have like a message that greets the user for the landing page
    - Customer likes the landing page (plus)
- Customer asked how we are going to implement some sort of balance associated with our account for a particular purchase
- Customer/Mentor proposed a way in of how we are going to use the website for ticket trading and all that
    - Customer likes the idea and proposed that we could have like documentation/help page to help the user with the questions they could have
    - Maybe also add in like a rules and regulation page (Terms and Service)
- Asked profile page and how is it going to look and difference between **Sellers vs Buyers**
- Customer asked that we could add a filter option in order to view what exactly a certain buyer is looking for
    - Asked question about whether seller is able to choose whom to sell the ticket to if there are multiple users proposing to buy a ticket that is of the price the seller is willing to sell at
        - Customer wants the seller to be able to choose

3. Implementation

3.1. Client

- We currently have a landing page that essentially establishes what exactly our web application or platform is used for giving our users/customers a general overview as to what we exactly do, and we've implemented the front end using HTML, CSS, JS, React, and Tailwind
- We've also built a login page, alongside a sign up page, and the dashboard once the user has successfully logged into their own individual account. We've decided to now work on the individual widgets on the side bar, and hopefully link each page to each other once we've successfully completed all the side bars
- One of our members is currently working on the myTickets page, and we're focusing on curating more interactive content soon for our users in the distant future

3.2. Server

- Our server connection is going to be done using Mongoose
- We will communicate with MongoDB Atlas using Mongoose, which we can use to set up the connection and then modify, insert, or delete anything in our database

- Using these methods built into Mongoose, we can create our own functions that push or pull information specific to our database, which can then be used by the frontend

3.3. Database

- Currently we are using MongoDB Atlas in combination with Mongoose
- MongoDB Atlas means our database is on the cloud, so we can simply connect to it and have it handle all the data
- We currently have tables for users, tickets, offers and bids
- The database is currently set to accept a connection from any ip address as long as it uses a connection string with the appropriate user and password, which would then give the connection admin privileges

3.4. Challenges & Risks

- Mongoose seemingly recently had an update, and as such almost all examples online use some deprecated features or differently formatted code
- Most help for setting up Mongoose with MongoDB refers to a local database, and not MongoDB Atlas, which has made it more difficult to find how to properly set it up since they are used differently
- One of our group members is still in the process of understanding React, so it has somewhat hindered our progress in one aspect; however, he's making progress towards completing that, and it's just been a learning experience, but we are steadily moving forward

4. Evaluation

4.1. Functional requirements

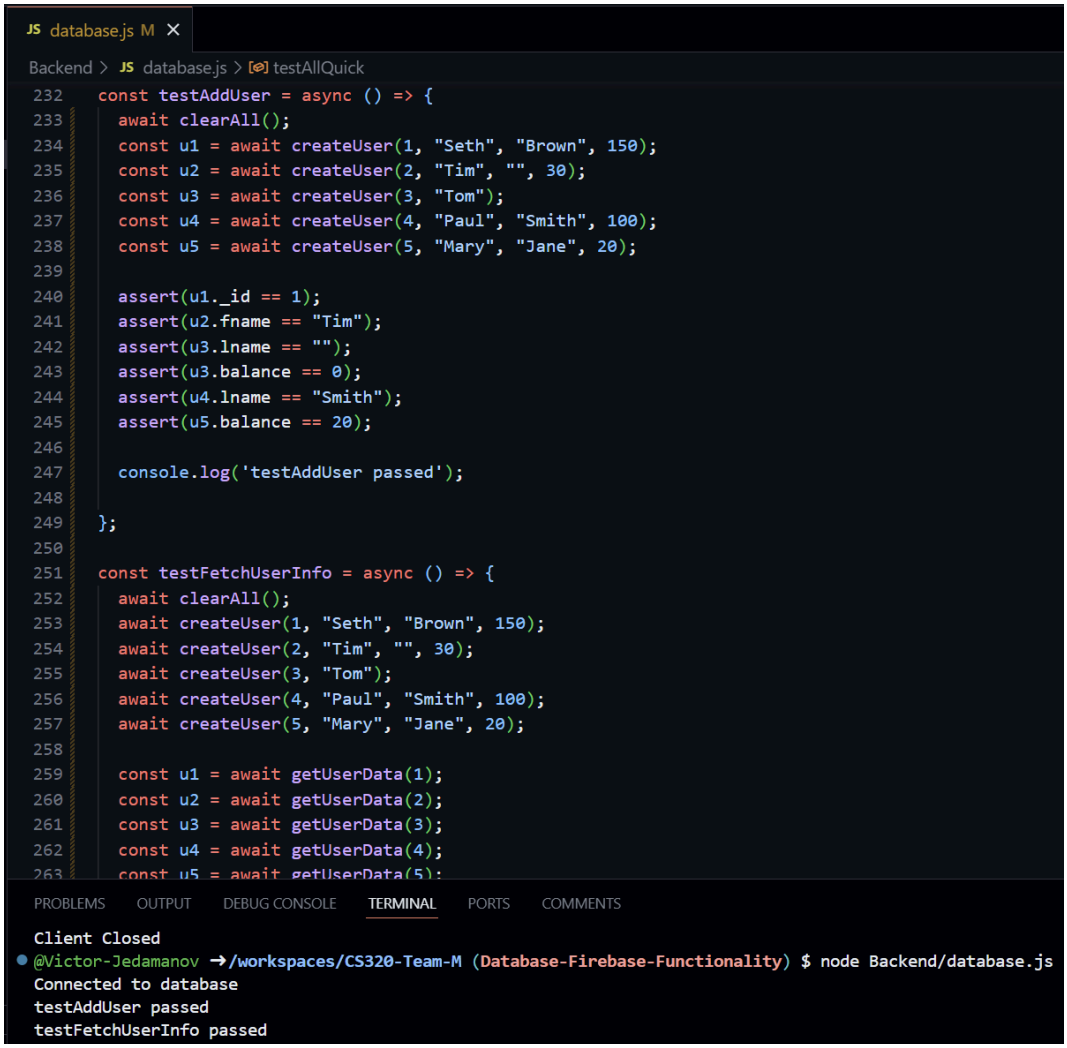
- Frontend
    - User acceptance tests:
        i. Testing if a user is able to **create a new account**:
            - When the user first opens the web application, they should see the landing page, and after viewing the functionalities and main purposes of our website and what it does, then they should be able to click on an area where it'll ask the user to either login or sign up for an account, and if it's a new user, then they should be prompted to click on the new user button, then they'll be redirected to a sign up page, and then the user will enter an email address, in addition to a username and password, and once the user does that, and it fits all the criterion, then the user will now be brought back to the landing page, with a text that pops up saying that the account has been successfully created
        ii. Testing if a user is able to **login** after they have already created an account:
            - The user should now click on the login button, and input their email address in addition to their password, and if successful, and the email of the user is correctly associated with the password that the user had set during the sign up process, then the user will be redirected to the

homepage/dashboard of Ticket Trading, but if the fields are incorrect, then the user will be be prompted with an error message saying that the fields are incorrect, and to please try again

iii. Test if a user is able to **access myTickets** on the homepage:
    - The user once on the homepage, has an assortment of different buttons displayed on the side, and one of the interactive buttons on the side is a button “myTickets” which essentially displays all the tickets that are currently associated with that particular individual that is currently listed in our database pertaining to that user; however, in order to display that ticket, the user has to insert the information associated with that ticket, but if the user bought the ticket from another user on the platform, then once the transaction has completed, then that particular ticket will be in the user’s myTickets too.

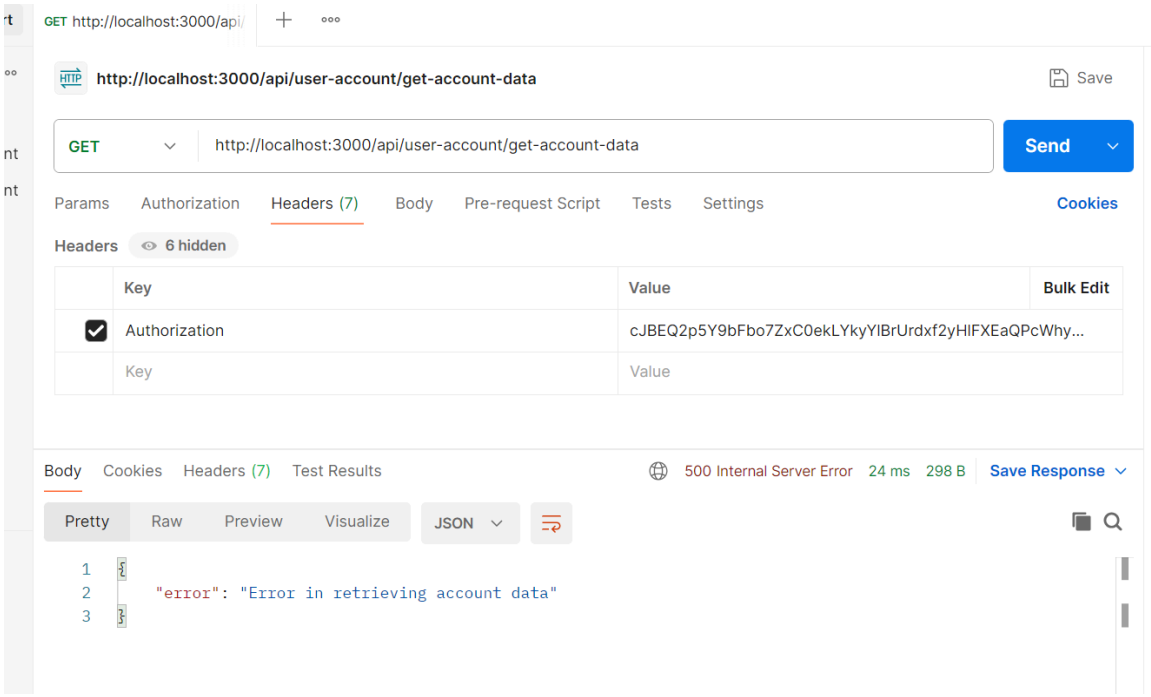
- Backend
    - We are planning on using MongoUnit as our unit testing framework
    - MongoUnit is designed to test that data in our database is what we’d expect it to be
    - Essentially, it allows us to test the functions that we’d make for the frontend using Mongoose and make sure they’re working properly

- Unit Testing



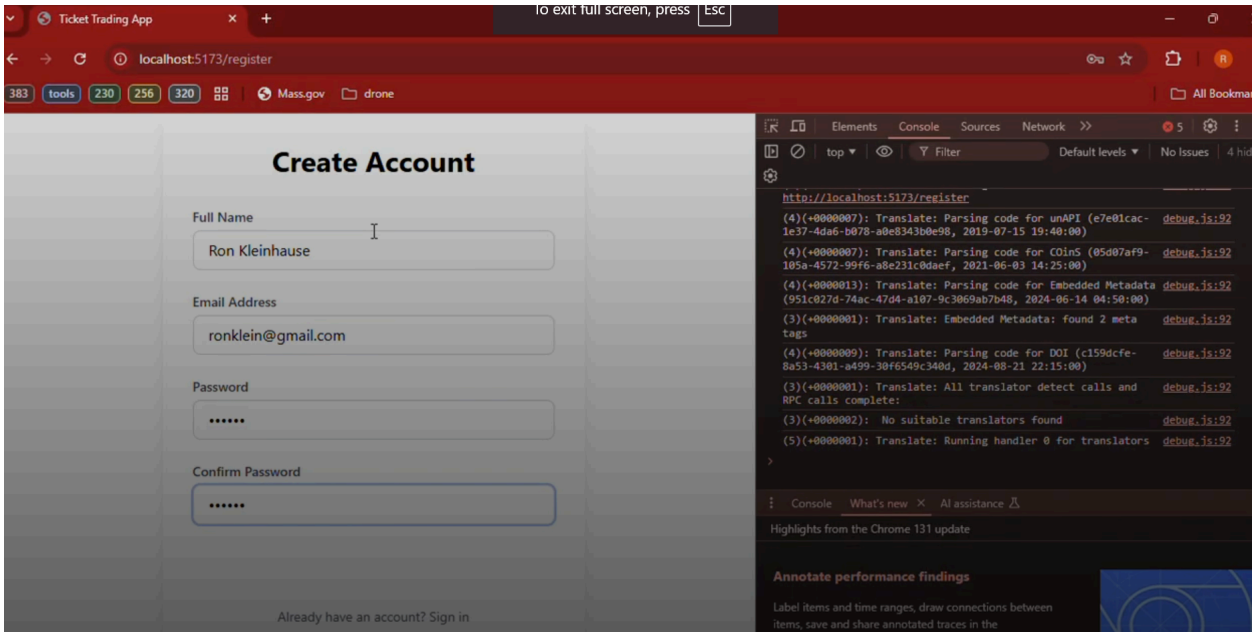
Testing database.js functions with custom testing utility. Assertions are made on various returned values from the database.

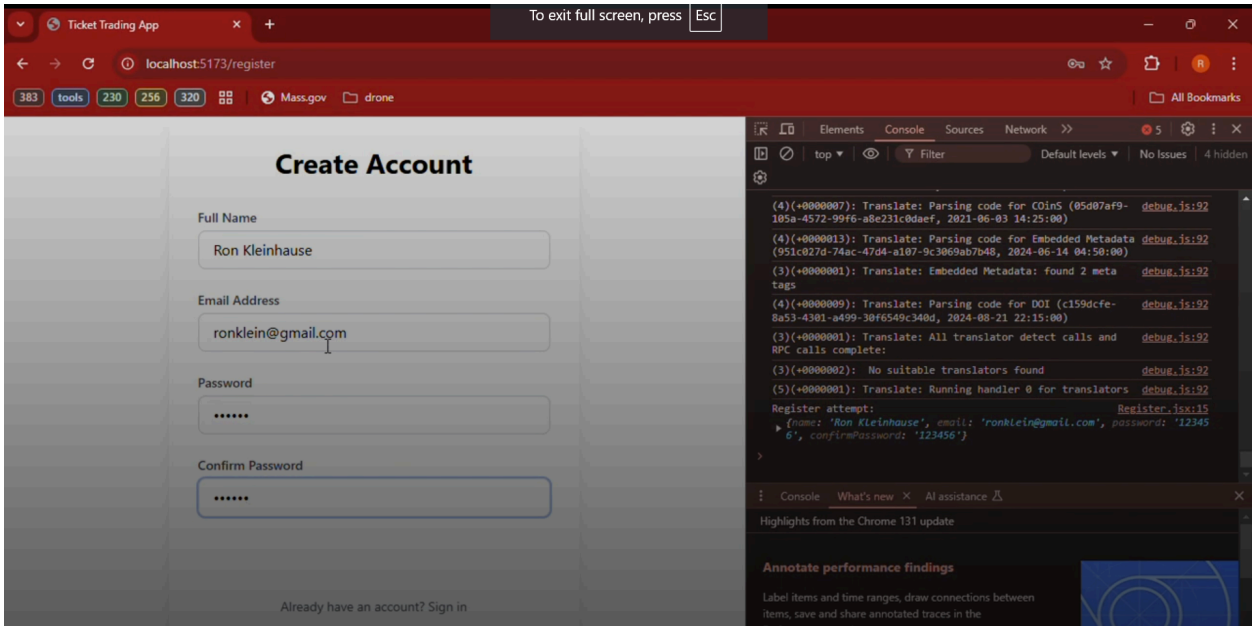
- Integration Testing

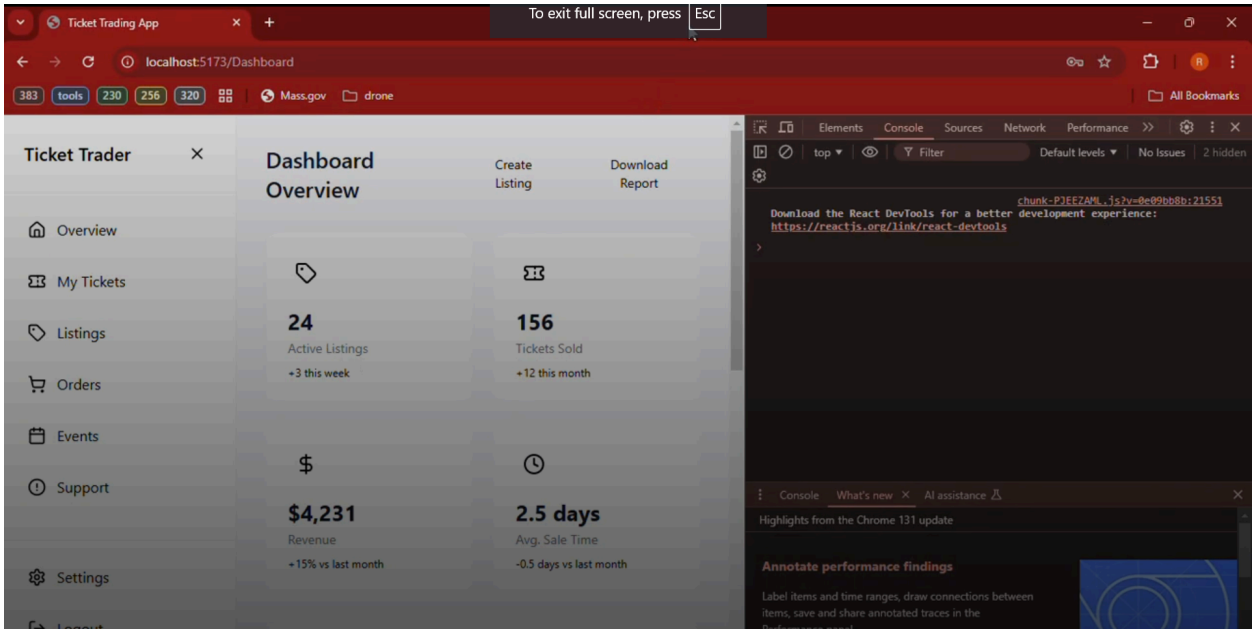


Shows API returns error when token is invalid.

- User Acceptance Testing







Shows Create Account page works.

4.2. Non-functional requirements

- Usability
  - The UI will be made to be intuitive (i.e. the main button will be larger, icons will be clear in their purpose)
  - We can make the webpage squash and stretch appropriately to fit different screen sizes

        - We can add color palette options for users, allowing them to use colors that are easier on their eyes if need be
- Data integrity
    - We can ensure the integrity of data by doing checks whenever something is going to happen to the database
    - This means if we are inserting data, we can check if it's the appropriate type
    - Also, if we are deleting data, we can check if it's being referenced somewhere else in the database and prevent the deletion if there is any data that depends on it

4.3. Challenges & Risks

- It's possible that our user acceptance tests would end up biased, and that we'd be testing things that we expect to work. This would mean that we wouldn't actually be finding vulnerabilities, just confirming what we already knew, which wouldn't be helpful in the long run
- Adding features that increase usability will be very time consuming, while at the same time not a high priority, making it less likely they'll be as polished as other parts of the project
- Ensuring data integrity with all these checks would increase the amount of computation we do every time we do anything with the database, which could cause issues if we have a lot of concurrent users

5. Discussion

All features aren't fully implemented as the backend was not fully integrated with the frontend, so backend functions aren't being called by the frontend for most things

- Create user accounts
    - The backend has functions that add associated information in the database. These functions fully work as intended.
    - The backend also uses firebase to create authentication tokens for each account, giving us a unique identifier for each user
- Log in/ Log out
    - The frontend has implemented a login page, though it is not connected to firebase for logins to actually occur
- Letting users to auction tickets
    - The backend has this fully implemented as a function
- Letting users to bid on tickets
    - This is also fully implemented as a function in the backend
- Letting event organizers/venues adding events to the system
    - This idea was changed as the project continued, so know users who created tickets are able to modify events attached to said ticket. The backend functions for creating tickets include event names and dates as an attribute, although they are not required fields
- In app "wallet"
    - The wallet was mostly ignored, as actually integrating our project with something like paypal would cost us money. Instead, our database simply stores each user's balance, and the backend has several fully implemented functions to handle this.
- Exchanging the tickets between accounts.

- Ensuring full ticket transfers didn't make it into the project as this would change between different events. Having physical tickets would make it impossible, other than possibly marking it as "transferred" in the backend, and different digital tickets might have their own methods for exchanging tickets. This made this a low priority feature to focus on.
- Also, allowing for orderbook matches, where a ticket offer is matched with an appropriate bid, is not currently implemented in our endpoints
- On the frontend, an orders page is made that would allow for tracking tickets transfers or managing offers/bids if integrated with the backend

Frontend Feature Completion Status:

- Core pages implemented: Home, Login, Register, Dashboard, Marketplace, MyTickets, Orders (100% complete)
- Navigation and routing system implemented (90% complete)
- Responsive design implementation (85% complete across all pages)
- Component reusability: Created 15 reusable components including modals, cards, and form elements
- Real-time updates and state management (40% complete)

Backend Completion:

- Database is fully implemented, with all functions for data modification and retrieval complete
- Endpoints are mostly implemented, only requiring integration with frontend for use
    - The check for matching offers to appropriate bids has not been implemented
    - A few endpoints for modifying data haven't been implemented yet

UI/UX Achievements:

- Successfully implemented consistent design language across all pages
- Created responsive grid layouts for ticket listings
- Implemented advanced filtering and search functionality
- Developed interactive modals for ticket details and creation
- Built intuitive forms with validation for user input

Frontend-Backend Integration:

- Developed placeholder data structures to continue frontend development while backend APIs were being built
- Implemented error handling patterns for API integration
- Built loading states and error states for async operations

Authentication Integration:

- Successfully integrated Firebase authentication for user management
- Implemented protected routes and authentication flow
- Created token management system for API calls

Known issues:

- The frontend and backend aren't fully integrated
- Several user or ticket details, like event dates and names, don't have pages made for their modifications
- Most, if not all, buttons on the frontend don't navigate to other parts of the webpage

Challenges:

- There were many issues when trying to work with firebase, which made it difficult when we tried to integrate it into our project
- We used Mongoose for our database management, and its documentation is extremely unclear. Also, most online forums that discuss the module reference deprecated code, which was unhelpful.
- For the frontend handling real-time updates for ticket status changes synced with database and backend
- Coordinating frontend-backend integration with different development speeds

If we had 2 more weeks:

- We'd focus on integration between the frontend and backend so that our endpoints can actually be used. This would also allow our frontend to be more dynamic as it could display actual used data
- We'd implement proper error handling and loading states across all components to provide better feedback during user interactions and API calls, making the application more reliable.
- We'd complete our endpoints and add full bidding functionality such that offers are matched with bids properly

Some challenges or risks that could come from this could include that integration will show us errors we may have overlooked due to not being fully implemented, which would require us to make changes to the backend along with the frontend. It could also be difficult to make the frontend display live updates of information from our database, as we haven't tested that out previously.

Final deliverables video:
https://drive.google.com/file/d/1ou_xIZTg72MijrjaHomDALr1kP_KVly0/view?usp=drive_link