TrustTheTickets.com
Detailed Design Document

TRUST THE TICKETS.

Initial Draft Date: October 9th, 2017

Final Revision: November 22nd, 2017

Authored by:

Anthony Orio

Christopher McKane

Curtis Baillie

Derek Gaffney

Jon D'Alonzo

Thomas Harker

https://github.com/JonDalonzo/Senior-Project

Table of Contents

Project Description

Project Introduction

In today's sports-saturated culture, fans are becoming more and more dedicated to their favorite sports teams and, as a result, are constantly looking for the best ways to find tickets to games. The sports fans in the Philadelphia area are no exception, and are in many ways far more outspoken about their sports teams than other fan bases around the United States.

Currently, there are a few high-profile ticketing sites like Ticketmaster, StubHub, and others, where fans can go to buy and sell tickets - but many of these sites impose hefty service fees (anywhere between 25% - 35%), taking a sizable chunk out of transactions between local sports fans. These sites are also very complex by nature, which can make it very hard to find the event you are looking to purchase tickets for.

This is where Trust The Tickets comes in. Fans of the Philadelphia 76ers can utilize TrustTheTickets.com to buy and sell their tickets to other fans. One of the most important aspects of the website design involved ease of use. Fans can find the event they are looking for within 3 clicks or less using one of the two search methods – Search by Calendar, or Search by Team. Trust The Tickets also charges less than half the fees of other popular websites (free for sellers, 10% processing fee for buyers). The fees and ticket attributes are always shown to the buyer clearly, alleviating much of the stress and confusion that comes with purchasing tickets online. The transaction process is simple, elegant, and painless.

The goal of this project is when the next time fans are looking for seats at a 76ers game comes around, they can skip Stubhub and go straight to TrustTheTickets.com.

Use Cases

1) Seller use cases
    a) Seller creates account
    b) Seller views past transactions (sales)
    c) Seller starts listing of tickets
        i) Provides quantity of tickets
        ii) Provides section, row, seat number

        iii) Provides quantities that people can buy at a time (multiples of 2, all tickets together)
        iv) Provides disclosures on tickets (obstructed view, no alcohol section, wheelchair accessible, etc)
        v) Provides comments on tickets (Early entry access, Aisle seats, concession credit, etc)
        vi) Sets pricing for tickets (all tickets priced the same amount)
        vii) Uploads PDF files
    d) Seller modifies listing of tickets
        i) Deactivates listing
        ii) Change price of listing
        iii) Change quantities of tickets that can be purchased in listing

2) Buyer use cases
    a) Buyer Account Functions
        i) Buyer creates account
        ii) Buyer modifies account
        iii) Buyer deactivates account
        iv) Buyer views past transactions
    b) Searching functions
        i) Buyer searches for a specific game
        ii) Buyer searches for games a specific team is playing in
        iii) Buyer searches for best value (price)
        iv) Buyer wants to browse
    c) Buying functions (Once on event listing page)
        i) Buyer filters tickets by quantity available
        ii) Buyer filters tickets by aisle seats only
        iii) Buyer filters tickets to exclude obstructed view seats
        iv) Buyer filters tickets by handicap accessible only
        v) Buyer filters tickets by zone
        vi) Buyer filters tickets by section
        vii) Buyer buys subset of tickets in a group
        viii) Buyer buys all tickets in a group

Exclusions

The following items have been explicitly set outside of the scope of this project:

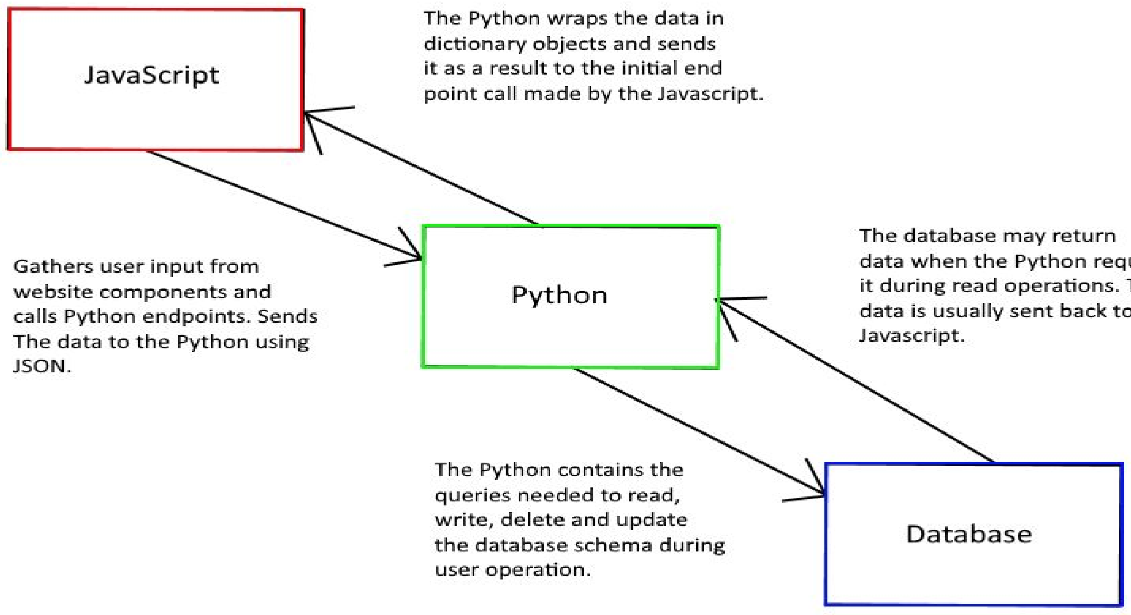
1. TrustTheTickets.com is currently only set up to sell basketball tickets for the Philadelphia 76ers for their home games at the Wells Fargo Center.
2. We are doing "simulated" transactions (no funds are being transferred). This is due to time constraints.
3. We are not using “disclosures” or “comments”. We are only doing extras (early entry, aisle seat, handicapped accessible).

4. We are not doing an exact representation of the Wells Fargo Center. (We are not doing a 1-1 seat, row, section mapping). We have a test version of the stadium with the upper and lower sections and test row and seats.
5. Users are expected to provide correct ticket information such as Section Number, Row Number, Seat Number and any extras such as Early Entry, Aisle Seat and Handicap Accessible ticket features.
6. Sellers are expected to provide valid tickets, as we do not have a method to validate the tickets. In the future, we will need to take down the seller's payment information in case of invalid tickets, so that they can be charged a fee equivalent to the cost of the buyer's replacement tickets.

High Level Design

Trust the Tickets is separated into three key components which interact with one another. The three components are the JavaScript frontend, Python backend and the MySQL Database. The JavaScript portion is tasked with rendering the web pages for the user and gathers user input to send to the Python, as well as handling front end state. The user input enters through the medium of buttons, text fields and other react-bootstrap components which are stored in various states and data variables. This data is transferred via endpoints deployed in the Python code (see endpoints section below) to the back end. All of the data that is sent from the javascript front end is wrapped in JSON before being directed through endpoint calls. The Python Flask server listens for any calls made by the Javascript and unwraps the JSON to perform the desired functionality, such as user login or retrieving data from the database for example. A Python class called "Back_Door" contains all of the endpoints the Javascript can use to interface with the basck end. The Back_Door class interacts with another Python class called SqlHandler, which is purposed to encapsulate all of the interaction between the database and the Python back end. SqlHandler contains functions that handle static and dynamically built SQL queries for updating, retrieving, writing and deleting from the Database. When these database operations need to occur during normal user interaction with the website, the SqlHandler executes the queries defined in the class to access the database. The database contains all tables deemed necessary to support the data model for website operation (see database section below).

The following image visually describes the flow of operation between the three components listed above:


JavaScript
The Python wraps the data in dictionary objects and sends it as a result to the initial end point call made by the Javascript.
Gathers user input from website components and calls Python endpoints. Sends The data to the Python using JSON.
Python
The database may return data when the Python requests it during read operations. This data is usually sent back to the Javascript.
The Python contains the queries needed to read, write, delete and update the database schema during user operation.
Database

Front End Design

The front end design section describes the pages of the website and the functionality of each of those pages.

1. Landing Page
    - Front page of website with logo and slogan
    - Top navbar for navigation to other pages
    - Information about what/who Trust the Tickets is for
    - Scroll down for a selection of different ticket choosing options
        - Search an event in a calendar
        - Select an event based off of the opponent
2. Calendar View
    - Show all upcoming games on a monthly calendar
    - Right hand panel shows information for each game for given month
    - The list of events on the calendar is from the /get-games-with-details endpoint
    - Selecting an event on calendar or right-hand panel redirects you to a page for purchasing tickets for an individual event (Arena view)
3. Teams View
    - Displays every opponent of the 76ers when horizontally scrolled
    - Can select a team logo
        - Teams pulled from endpoint: /all-teams
        - Available games appear below logos by endpoint: /games-by-team
        - By choosing a game option it will direct you to the Arena View
    - You are able to select another game any time before selecting a game option
4. Arena View
    - Shows seating chart of arena with available tickets in a panel on the right
        - Ticket data initially pulled from the /get-cheap-ticket-any-section
    - Allows for filtering of queried tickets
        - Filtered searches retrieve tickets from the following endpoints: /pick-ticket-filter, /search-tickets-with-filter, /search-tickets-in-zone-with-filter, /search-tickets-in-section-with-filter, /get-tickets-and-sections-by-price
    - One or more sections and zones are selectable.
        - Selecting a section updates the side ticket panel.
    - Tickets in the ticket panel are selectable
        - Selecting a ticket opens new page where you can purchase ticket
        - Upon transaction completion, tickets transferred via email
5. My Account
    - If not logged in, redirects to Login page.
    - Login data sent to endpoint: /login

- If logged in, you can view account info
    - Data pulled from endpoint: /my-account
    - View purchase history
    - View listing history
    - Create a Listing
    - Cancel or edit tickets currently listed

6. Create Listing View
    - Product input page
        - Require user login
        - Collect all relevant ticket information from seller
            - Event
            - Event date
            - Section, row, seat number(s)
            - Price
        - Once information filled out, data sent to /validate-ticket-info endpoint for saving/verification
        - If valid, data saved using endpoint: /send-listing-data
7. My Listings View
    - Ticket listings will be displayed with status at the top
        - Listing data pulled from endpoint: /your-listings
        - In progress (in blue), or <the date purchased> Completed (in grey)
    - Edit button
        - Allows the ticket price to be modified
        - New data sent to endpoint: /update-listing
    - Cancel Listing
        - Removes listing from database using endpoint: /cancel-listing
    - Details are given within the ticket window
        - The opposing team, the date and time, the section, row, seat(s), the ticket prices, amount of tickets available to sell as a group, # of tickets listed, # of tickets remaining (that have not been sold)
8. Purchase History View
    - Displays each ticket package purchased
        - Purchase data pulled from endpoint: /your-purchases
        - Contains the teams playing, the date of game, the section, row and seats of the ticket package, and the price
    - The logo is the opposing team is shown on the left side of individual section
    - Order Details button
        - Breaks down the amount the user paid
            - Shows how many tickets were purchased and at what price, the processing fee, and the transaction total
9. Ticket Selection Modal
    - Displays available tickets in selected group
        - Uses endpoint: /get-tickets-for-group

    - Allows user to select tickets to purchase
    - Once selected these tickets are on hold for that user for some time frame
        - Accomplished using endpoint: /hold-tickets
10. Checkout Page
    - Displays selected ticket information and price along with relevant fees
        - Uses endpoint: /get-fees
    - If user completes transaction, data added to database using endpoint: /insert-transaction
    - If user hits back button the tickets are unlocked using endpoint: /unlock-tickets
11. Login
    - Login with email, password
    - Uses /login endpoint for credential authentication
12. Registration
    - Create account with email, password
    - Country and state info pulled from endpoints /get-country-names and /get-country-states respectively.
    - Uses /register endpoint to send credentials to database
    - Backend sends authentication email to the provided email address
    - Clicking link sends user to site with unique query param identifier
        - This query param is POSTed to /registration-confirm to confirm account registration
    - A page confirming account registration is then displayed to the user

Back End Design

Descriptions of the backend endpoints for our Flask server. (Note: all input and output in JSON format)

1. Send Tickets PDF (/send-tickets-pdf)
   - Input: token, ticketIds
   - Output: success value, side effect of mailing ticket pdfs for the corresponding ticket id's
   - General Algorithm: Send Tickets Pdf receives a token from the front end that corresponds to an account_id. The account_id is looked up using the jwt_service.get_account method, as well as the email using SqlHandler.get_user_email. Then the ticketId's are passed off to the PdfWorker object which using the S3Worker retrieves from the S3 bucket the pdf files for each corresponding ticket_id and bundles them into one pdf file. The pdfWorker returns the resulting pdf file. The method then emails out the pdf to the account's email, returning success if it is successful.
2. Token refresh (/token-refresh)
   - Input: The token of the current user.
   - Output: An error if there is no content, a false authentication if there is no token, or the current token and a true authentication.
   - General algorithm: Refresh the token so long as content is there and the token is not null.
3. Register (/register)
   - Input: Email address and password
   - Output: If successful, entry added to account_registration table and confirmation email sent to provided address. Outcome of registration attempt returned to front end.
   - General algorithm: If there is not a pre-existing account with the same email address, insert a record into account _registration with: a unique UUID registration_code, the provided email address, and provided password. Send a confirmation email to the provided address with the registration code as a query parameter.
4. Confirm registration (/registration-confirm)
   - Input: Unique account registration code
   - Output: If successful, entry added to accounts table. Outcome of registration confirmation returned to front end.
   - General algorithm: Read the query parameter from the POST request. If the parameter matches an UUID in the account_registration table, the corresponding email address and password are added to the accounts table and registration is confirmed.
5. My account (/my-account)
   - Input: The token of the current user.
   - Output: The account information of the current user.
   - General algorithm: Gather the information of the user currently signed in.
6. Get Games With Details (/get-games-with-details)
   - Input: The start and end date of desired tickets.
   - Output: Every event within the start and end dates, with details for the events

- General algorithm: Get each game that falls between the start and end dates that were given. Uses the SqlHandler get_games_with_details method.

7. Get events (/get-event)
    - Input: EventID.
    - Output: The home team, away team, and the title in this sort of format "<home team> vs. <away team>".
    - General algorithm: Gets the output data from the database.
8. Search for Tickets with a Filter (/search-ticket-with-filter)
    - Input: eventId, minPrice, maxPrice, earlyAccess, aisleSeat, handicap, desiredNumOfTickets
    - Output: The collection of tickets that meet the filter criteria (tickets, sections)
    - General algorithm: Retrieves all tickets and the sections they are in that meet the same filter restrictions that are given in the input.
9. Search for Tickets in Sections with a Filter (/search-tickets-in-sections-with-filter)
    - Input: eventId, sections, minPrice, maxPrice, aisleSeat, earlyAccess, handicap, desiredNumOfTickets
    - Output: The collection of tickets that meet the filter criteria (tickets, sections) that are in the sections provided
    - General Algorithm: Retrieves all tickets and the sections they are in that meet the same filter restrictions that are given in the input and are in the provided sections.
10. Search Tickets in Zone with Filter (/search-tickets-in-zone-with-filter)
    - Input: eventId, section_type_id, minPrice, maxPrice, aisleSeat, earlyAccess, handicap, desiredNumOfTickets
    - Output: The collection of tickets that meets the filter criteria in the zone provided
    - General algorithm: Retrieves all of the tickets associated with the zone that was chosen by the user and also meets the filter criteria.
11. Get Tickets and Sections By Price (/get-tickets-and-sections-by-price)
    - Input: eventId, aisleSeat, earlyAccess, desiredNumOfTickets, priceMode, priceRange[min], priceRange[max]
    - Output: Retrieves all tickets and the sections they are in that meet the same filter restrictions that are given in the input.
    - General Algorithm: Retrieves all tickets and sections they are in that meet the same filter restrictions that are given in the input and are also between the priceRange provided. This method gives the opportunity to search for the most expensive tickets, least expensive tickets, or any tickets within a specific range, depending on the price mode value given.
12. Login (/login)
    - Input: Email address and password
    - Output: Outcome of login attempt returned to front end
    - General algorithm: Run select query on accounts table using credentials. If one result returned, user login succeeds.
13. Games (/games)
    - Input: None.
    - Output: The collection of games.

- General algorithm: Get the list of teams for each upcoming game.

14. All teams (/all-teams)
    - Input: None.
    - Output: The list of teams.
    - General algorithm: Get every team that is in the database.
15. Games by teams (/games-by-team)
    - Input: Team id.
    - Output: The list of games.
    - General algorithm: Get every game that is associated with a given team.
16. Your listings (/your-listings)
    - Input: Account Id
    - Output: Collection of user ticket listings and a Boolean to say it was an authenticated pull. Returns false if an account is not logged in and the pull was not authenticated.
    - General algorithm: Uses the unique account token to get the account ID then searches the database for all of that user's ticket listings.
17. Your Purchase (/your-purchase)
    - Input: Account Id
    - Output: A collection of ticket information that the user has purchased through that account. Returns false if an account is not logged in and the pull was not authenticated.
    - General algorithm: Retrieves all of the information linked to the account Id that pertains to all of the information of the tickets the user has purchased.
18. Update Listing (/update-listing)
    - Input: User Token, groupId, newPrice
    - Output: A collection of the user listings that have the new and updated price added to the database. Returns false if an account is not logged in and the pull was not authenticated.
    - General algorithm: Retrieves the user's listings from the database using the account Id from the token then uses the groupId to find the tickets that need to be updated with the new price.
19. Hold Tickets (/hold-tickets)
    - Input: User Token, TicketIds
    - Output: Result of holding the tickets, if it was successful or not, and a timer value.
    - General Algorithm: Uses the user's unique token to assign the locked tickets to their account and locks those ticket Ids passed into the method. The tickets are locked for the given timer value that is also passed into the method.
20. Cancel Listing (/cancel-listing)
    - Input: User Token, groupID
    - Output: A collection of the listings that were canceled by the method as well as the transactions that were associated with the listing. Returns false if an account is not logged in and the pull was not authenticated.
    - General algorithm: Uses the unique user token to get the account Id of the user that is canceling their listing. Then queries for the group Id of the listing associated to the account Id then removes them from the database.
21. Create groups (/create-groups)

- Input: Array of Tickets
- Output: json groups of tickets
- General algorithm: Takes an array of tickets and sorts them by the group_id, putting them into groups of tickets and returning them as a dictionary with the group_id as the key for each group of tickets.

22. Get Group Of Tickets (/get-tickets-for-group)
    - Input: groupId
    - Output: tickets
    - General Algorithm: Passes in a group id, the SqlHandler queries the database for tickets with the corresponding group_id.
23. Get dates of games (/get-game-dates)
    - Input: None.
    - Output: A collection of game dates that are stored in the database.
    - General algorithm: Queries the database for all of the game dates that are stored within.
24. Get opponent by date (/get-opponent-by-date)
    - Input: Game Date
    - Output: The name of the opponent that plays on the given game date.
    - General algorithm: Uses the game date to query the database so that the opponent name can be retrieved.
25. Get country names (/get-country-names)
    - Input: None.
    - Output: A collection of country names.
    - General algorithm: Queries the database for all of the country names stored within.
26. Get country states (/get-country-states)
    - Input: Country Id
    - Output: A collection of states that are linked to the given country id.
    - General algorithm: Takes the given country id and queries the database to get the name of the states that are linked to that country.
27. Get Fees (/get-fees)
    - Input: None.
    - Output: A collection of percentages that pertain to the fees of the website.
    - General Algorithm: Queries the database for all of the stored fee information.
28. Insert Transaction(/insert-transaction)
    - Input: User Token, Tickets, Commission, Tax, Subtotal, Total, Group Id, TaxPerTicket, CommPerTicket, subtotalPerTicket
    - Output: A success message
    - General Algorithm: Passes off the transaction data to the create_transaction method of the SqlHandler object to create a transaction in the database.
29. Unlock Tickets (/unlock-tickets)
    - Input: ticketIds
    - Output: success value
    - General Algorithm: Receives a list of ticket id's to unlock. The ticket id's are passed off to the unlock_ticket method of the SqlHandler object. That object executes a query to set

the ticket_status_id of those tickets back to 1 and the lock_account_id = null if the ticket id was currently 4 (locked).

30. Send Listing Data(/send-listing-data)
    - Input: pdfFile, json containing (sectionNum, rowNum, seatsInfo, ticketPrice, numberOfTickets, minPurchaseSize, gameDate, accountID)
    - Output: A success message that confirms if the listing was created correctly.
    - General Algorithm: Receives a pdf file containing pages for each of the tickets that are to be created and json data containing all of the information neccessary to create a new ticket listing. All of this data is passed off to the ListingCreator object. The listing creator component takes the listing data and executes the MySQL queries needed to build the listing in the database. The Listing Creator also passes off the PDF file to the PdfWorker object. The PdfWorker object splits the PDF into separate ticket pdf files and uploads those files up to our Amazon AWS S3 service using the python component S3Worker.
31. Validate Ticket Info (/validate-ticket-info)
    - Input: sectionNum, rowNum, seatNums, gameDate
    - Output: ticketInfoResults (locationResults, ticketListedResults)
    - General Algorithm: Receives location data for a new listing in the form of the section number, row number, and seat numbers. Passes the data off to the SqlHandler to verify that the section, rows, and seats exist, and then verifies that no tickets are currently listed for the same game date in any of those seats. The results for the valid location query are returned back as a dictionary called locationResults, and the results for the currently listed ticket query are returned as a dictionary called ticketListedResults.

Database Design

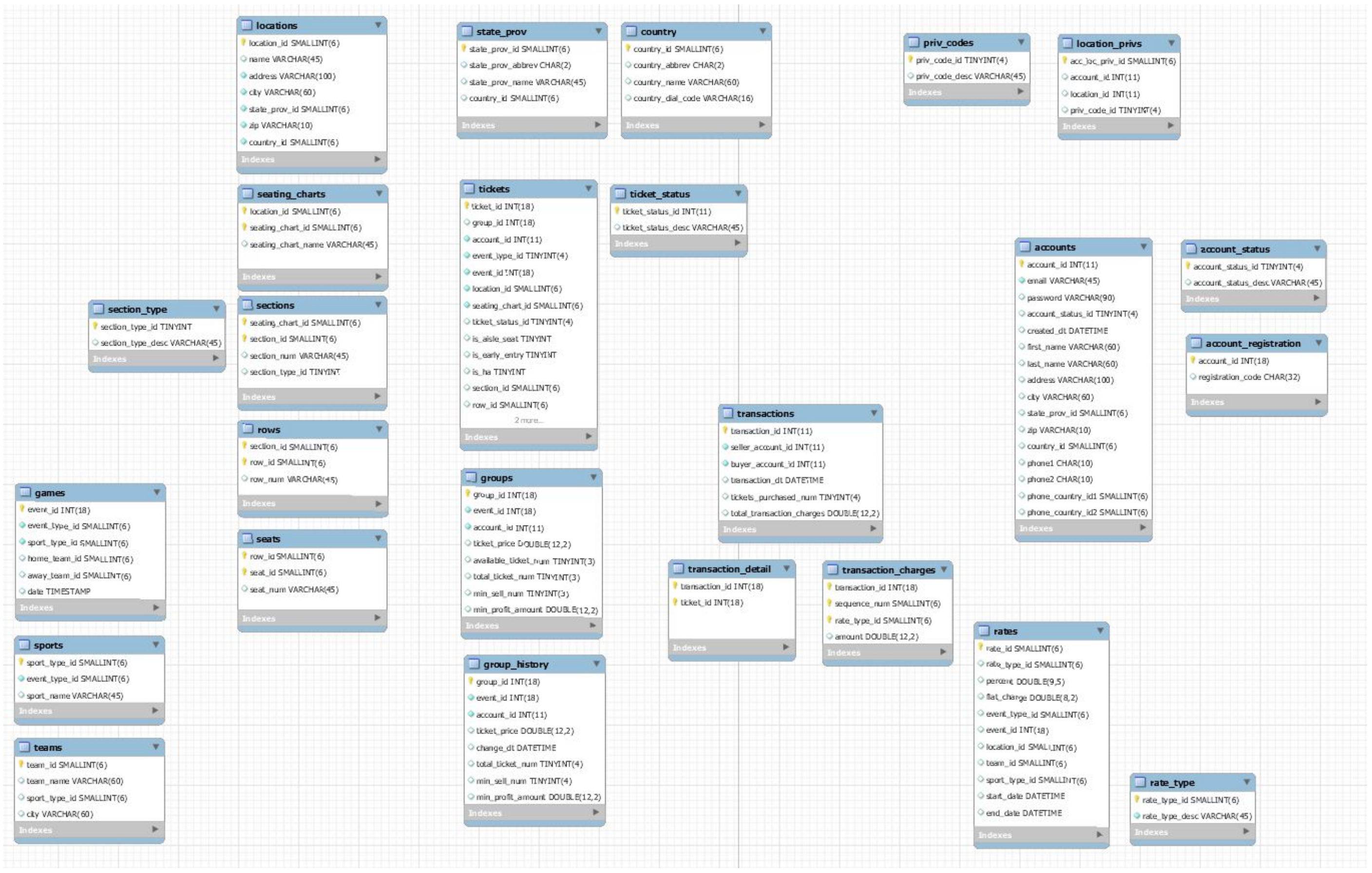
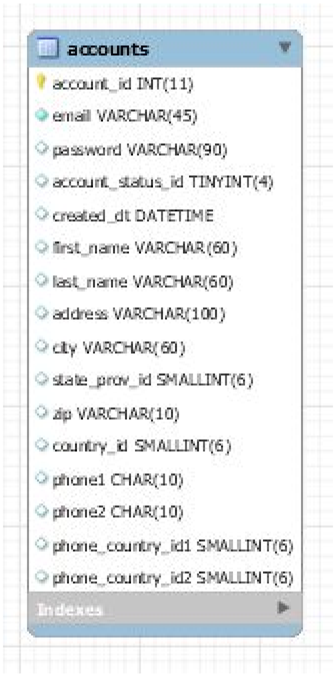


Figure 1 - Full database schema



Overview:

The database model is built around a few groups of entities. This section of the document will explain the relationships between these database entities.

Accounts:

The account table stores data pertaining to all users on the website. Both buyers and sellers will be account entities. Accounts are identified by their account_id.

The account_status_id column references a value from the account_status table which will store status values that will be used to modify accounts. The main use so far will be to have ‘unverified’ and ‘active’ values that will be used to differentiate between accounts that have just been made on the website from accounts that have been confirmed through the account confirmation process (verification through a link sent to an email address).

The account_registration table is a temporary table with two columns for the account_id and a registration_code. To verify their newly created account, a user has to click on the link emailed to them when they originally registered. The code appended as an argument to the link is the same as the registration code. Once the user has verified their email by clicking on their activation link, the corresponding row in the account_registration table is deleted and their account_status_id is updated in the accounts table.

Accounts contain detailed contact information including email, first name, last name, and address information. Notice that state_prov_id and country_id are primary keys referencing unique state and country values for the account in tables state_prov and country.

Phone_country_id1 and phone_country_id2 are country id’s that point to rows in the country table. This allows an account to have phones on file with country codes from different countries. The country id points to the country row the phone number is for, and there is a corresponding column in the country table that stores that country’s phone code.

Country:

The country table stores country values with a column for dial codes for those countries.

State_Prov:

The state_prov table stores state / province values for the United States and Canada.

Locations and Seating Charts:

Locations are considered any place that an event might be held. ***An event is anything you might sell tickets for***. Locations include stadiums, arenas, and other types of venues.

Locations have seating charts. A seating chart is a configuration of seating arrangements made for a specific one-time or recurring event. A location can, and must have many different seating charts. Examples of seating charts include the Philadelphia 76ers seating configuration in the Wells Fargo Center, or the seating configuration for a Flyers game in the Wells Fargo

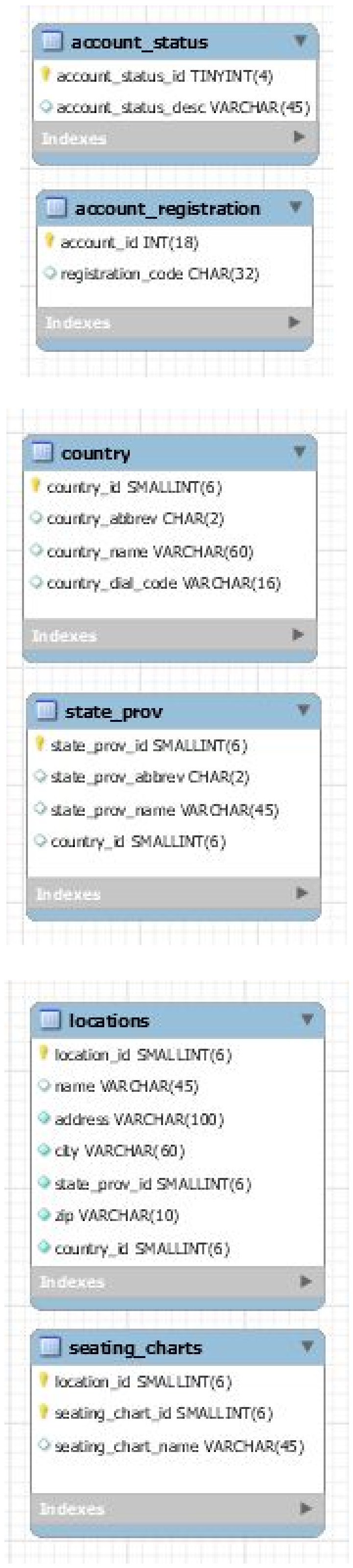
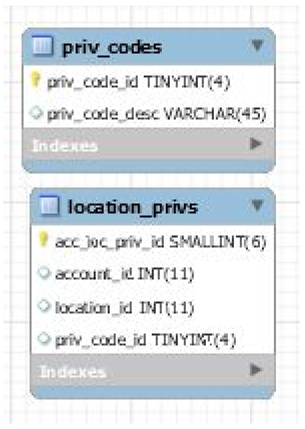


Figure 5 - Location-related tables

Center. This is a good example where one location (the Wells Fargo Center) has multiple seating charts for different events (Flyers and Sixers).

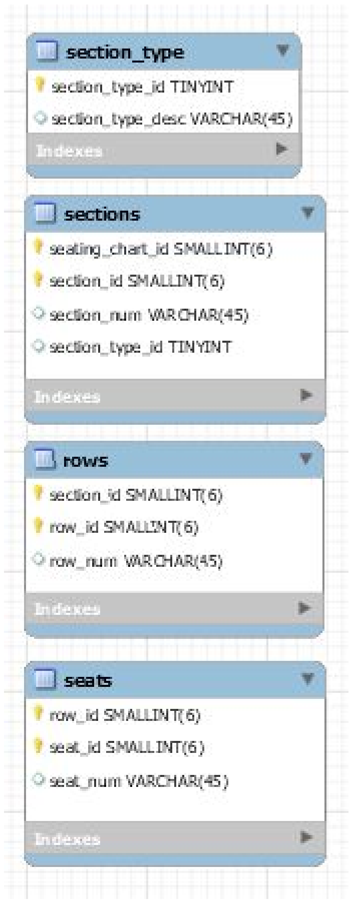
Seating charts have a reference to a specific location via the location_id, and locations have references to the state and country it is in via the state_prov_id and country_id.



Location_Privs:

Since accounts and locations have now been mentioned, location_privs can be explained. Since it is possible for accounts to have tickets for more than one location, it is important to be able to assign privilege codes to accounts to be able to keep track of their relationships with different venues. This is the reasoning behind having the location_privs and priv_codes tables.

The location_privs table keeps track of specific types of privileges that accounts can have for locations. Some examples of values include: ‘full-control’, ‘season-ticket-holder’, ‘view-only’. For this implementation, this table will most likely only store ‘full-control’ values, but if more types of account restrictions need to be implemented at a later date, this table of codes can be added to, and those codes can be assigned to an account-location pair in the location_privs table.



Sections, Section Types, Rows and Seats:

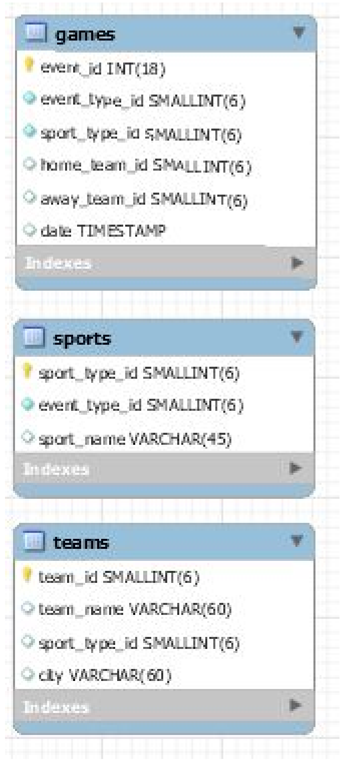
These tables contain rows of data that correspond to each unique section, row, and seat in the stadium. Every seat, row, and section has its own unique id, and a corresponding section_num (201A, 113, etc…). When a stadium is created the sections, rows, and seats that correspond to it need to be populated in these tables.

For sections, each section must have a section type (zone of the stadium), seating_chart_id (corresponds to a given seating chart configuration), section_num (client facing name of the section) and unique section_id (used within the application). Sections all can have rows. These rows are defined in the rows table with section_id’s corresponding to their particular section. The same holds for the seats in their respective rows.

Games, Sports, and Teams:

The current implementation of the website will sell tickets for sporting events, so these three tables are required. The games table holds all events that are considered sporting games, and each row has a reference to the sport the game is in. Since games are played between two teams, there are home and away team columns that refer to teams stored in the teams table. Those teams also have references to a sport in the sports table.

Every sporting event that a ticket is sold for will be contained in the games table. Having an event_type_id of 1 means that the event is a sporting event that will be stored in the games table. Other event_type_id's will not be implemented in the foreseeable future.



games
event_id INT(18)
event_type_id SMALLINT(6)
sport_type_id SMALLINT(6)
home_team_id SMALLINT(6)
away_team_id SMALLINT(6)
date TIMESTAMP
Indexes

sports
sport_type_id SMALLINT(6)
event_type_id SMALLINT(6)
sport_name VARCHAR(45)
Indexes

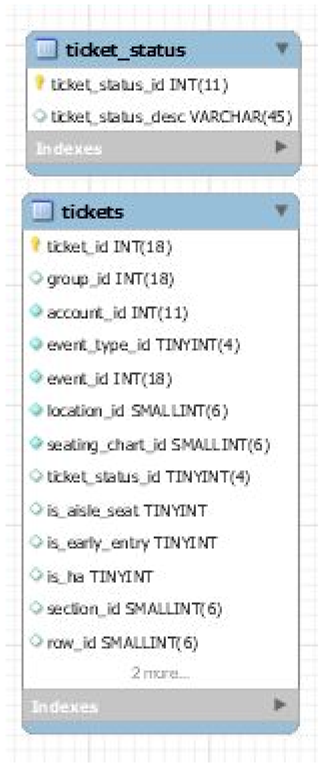
teams
team_id SMALLINT(6)
team_name VARCHAR(60)
sport_type_id SMALLINT(6)
city VARCHAR(60)
Indexes

Tickets:

The tickets table is one of the main entities in this data model. A row in the tickets table stores all identifying information on a ticket. A ticket is identified uniquely by its ticket_id. The group_id field maintains which group a ticket was uploaded into. More information on groups is included in the Groups section. The event_type_id of a ticket will be set to 1, indicating that all tickets are for sporting events (see section Games, Sports, and Teams for more info). The event_id of a ticket references the unique event that the ticket is for. Ticket_status_id references a row in the ticket_status table.

The ticket_status table stores different statuses that can exist for tickets: 'available', 'sold', 'cancelled', 'locked', etc. This status will change when tickets are uploaded, completed in a transaction, cancelled, or locked.

The tickets table column pdf_link was meant to contain a relative path to the pdf file for the ticket. This column is not currently implemented. Instead the ticket_id of a ticket is used as the unique identifier for ticket pdf storage. The account_id for a ticket refers to the account that uploaded the ticket.



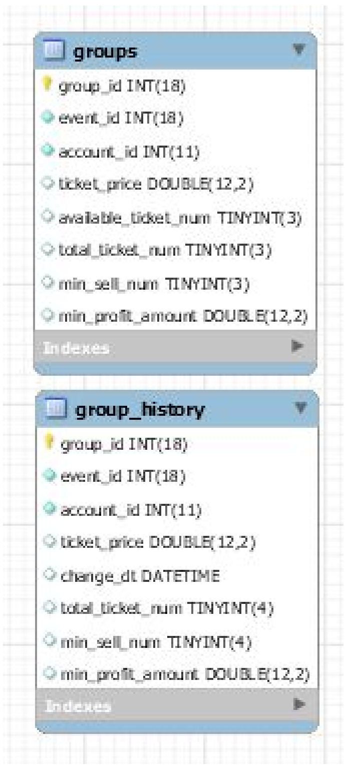
ticket_status
ticket_status_id INT(11)
ticket_status_desc VARCHAR(45)
Indexes

tickets
ticket_id INT(18)
group_id INT(18)
account_id INT(11)
event_type_id TINYINT(4)
event_id INT(18)
location_id SMALLINT(6)
seating_chart_id SMALLINT(6)
ticket_status_id TINYINT(4)
is_aisle_seat TINYINT
is_early_entry TINYINT
is_ha TINYINT
section_id SMALLINT(6)
row_id SMALLINT(6)
2 more...
Indexes

Groups:

When tickets are uploaded, they are almost always uploaded in groups of other tickets. For example, account gaffneyd4 uploads four tickets to the 76ers game in the same section same row with the same price. This is the most common case. It is for this reason that when tickets are uploaded onto the website, they are referenced in groups.

When tickets are uploaded, a unique ticket row is created for each ticket in the tickets table, and a single group row is created in the groups table for those tickets. The group row stores group-level information about all of those tickets. Ticket_price stores the price per ticket in the group (If 2 tickets are uploaded with a ticket_price of $50, each ticket costs $50 and the total cost will be $100). Available_ticket_num keeps track of the total number of tickets that are still available in the group. The total_ticket_num column is set once, and holds the number of tickets that were uploaded into the group. Min_sell_num stores the number of tickets that need to be sold for a transaction to take place (if 4 tickets are uploaded and the min_sell_num is 2, then a buyer can buy 2 of the 4 tickets, leaving 2 tickets remaining to be sold). Min_profit_amount stores a value that could be used a profit threshold, preventing transactions from taking place unless the seller were to make that threshold.

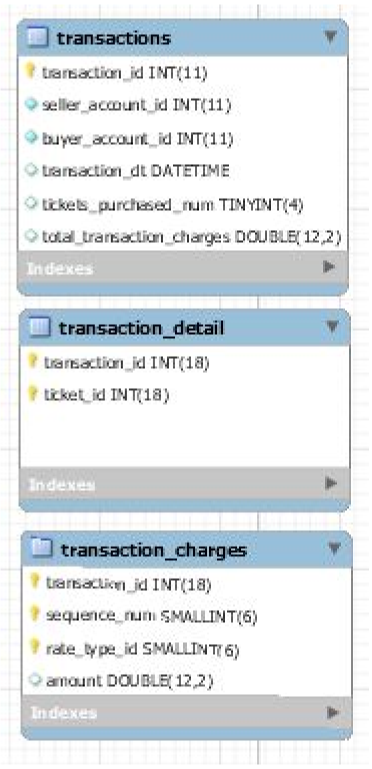


The group_history table stores “snapshots” of the state of groups. When a group is originally uploaded, a row will be inserted in the group_history with all of the same values as its corresponding row in the groups table, plus a timestamp column. Whenever a seller changes the information of one of their selling groups, i.e. min_sell_num, the changed row will be updated in the groups table and then a corresponding row will be inserted in the group_history table with a new timestamp.

This table will allow pricing trends to be tracked, and other statistics to be collected about how listings are being updated.

Transactions:

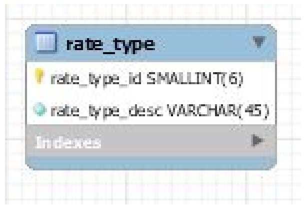
Transactions are between two accounts, a buyer and seller account. A transaction is created when a buyer initiates buying tickets listed by a seller. The transaction_detail table is the way to see which tickets have been transferred during the transaction. The tickets that have been purchased get added into the transaction_detail table, paired with the id of the transaction. The transaction_charges table contains rows that will sum up to the total_transaction_charges column for the transaction in the transactions table.



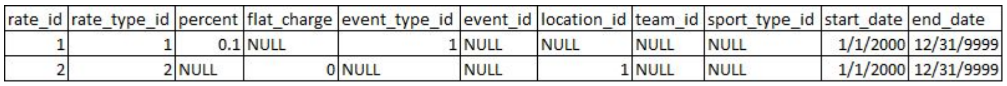
For each ticket in the transaction, there will be a sequence number paired with a transaction_id in the transaction_charges table. For those rows, different types of rates will be added into the transaction_charges table for each ticket. So if there are 2 tickets being purchased in a transaction with transaction_id ***x***, and there are two types of rates (the seller price and a tax rate), then there will be a total of four rows in the transaction_charges table for transaction_id ***x***. Rates are explained in the rates section.

Rates:

The rates table stores several rows with certain attributes that are compared against the attributes of tickets during a transaction. If anattribute applies to a ticket in the transaction, then the rate is added into the transaction_charges table for that transaction. Every rate has a description defined in the rate_type table, referenced by a rate_type_id.

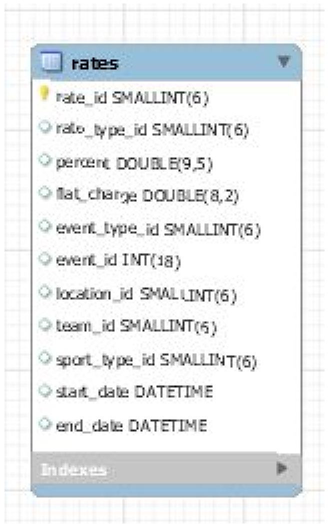


An example of two rates would be:



rate_id	rate_type_id	percent	flat_charge	event_type_id	event_id	location_id	team_id	sport_type_id	start_date	end_date
1	1	0.1	NULL	1	NULL	NULL	NULL	NULL	1/1/2000	12/31/9999
2	2	NULL	0	NULL	NULL	1	NULL	NULL	1/1/2000	12/31/9999

The first row (rate_id = 1) would be an example of a commission rate. This rate applies to all tickets that have an event_type_id of 1 (aka everything). All other criteria for this rate are null. The period that this rate is active is January 1st, 2000 to 12/31/9999 (forever). If the commission rate was to be updated, then a new row would be inserted into the rates table with the same criteria, with an update for the percent column, and the start and end dates would be different for the new row, and the end date for the first row would be updated to reflect the day the rate change would take effect.



The second row (rate_id = 2) is an example of a flat chart (in this case, $0) for a location with location_id = 1. This is an example of how certain locations, teams, sports, events, etc, could have unique fees assigned to them.

Conclusion:

This section of the design document is not a comprehensive data model specification document, but is rather meant to serve as an introduction to the general design and usage of various tables in the database. It is not intended to cover every use case or scenario, but hopefully paints a picture in general for how the database will be used to store and manipulate business data for the project.