TrustTheTickets.com  
Detailed Design Document

Initial Draft Date: October 9th, 2017

Last Revision: October 15th, 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

[Prototype Walk-through 3](#_Toc498883344)

[High Level Design 4](#_Toc498883345)

[Wormhole 5](#_Toc498883346)

[Use Cases 6](#_Toc498883347)

[Webpages/Navigation 7](#_Toc498883348)

[Python Microservices/Endpoints 9](#_Toc498883349)

[Database Specification 11](#_Toc498883350)

[Diagrams 16](#_Toc498883351)

[Implementation Goals by Mid-Assessment: 17](#_Toc498883352)

[Exclusions from Final Submission: 18](#_Toc498883353)

Prototype Walk-through

The first location we stop at is the landing page for Trust The Tickets. This page contains a navigation bar at the top, some text welcoming Sixers fans and two ways of searching for tickets. The first way is to view a calendar with every game listed, and choose which game you would like to view available tickets for. The second way brings up a list of all opponents and allows you to select an opponent, and then a game against that specified opponent. Once a game is selected, the user will be lead to another page to choose an arena section and browse ticket information. The navigation bar displays the name of the website and options to view a “Calendar” of games, view all “Teams” that the 76ers will play against, and review “My Account”. If we were to click the “Trust The Tickets” logo it would bring us back to this landing page.

Pressing the “Calendar” link (or “Search our Event Calendar”) loads a calendar with team logos and starting prices for each game. Once the user selects a game, they can click “See Tickets” to bring up a seating chart. This seating chart is a view of the Wells Fargo Center and allows the user to choose a section, which will then populate each group of tickets in that section into the panel on the right.

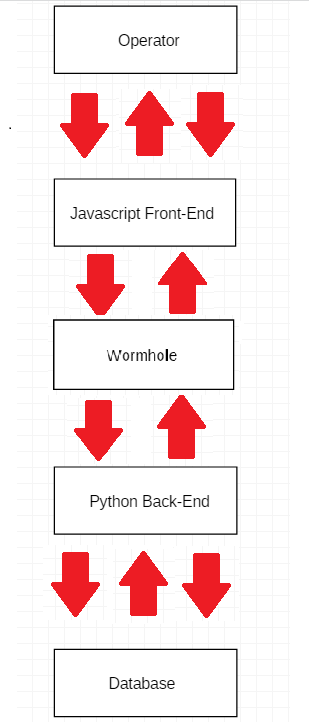
Choosing the “Teams” link (or “Search by Opponent” link below) loads an array of clickable team logos.

Pressing the “Pick Tickets” widget loads a new page that allowing the user to choose a section in which they would like to see the available tickets. This is where a picture of the Wells Fargo seating lives that will let the user click any section they decide to which will then populate each ticket in that location into the panel on the right.

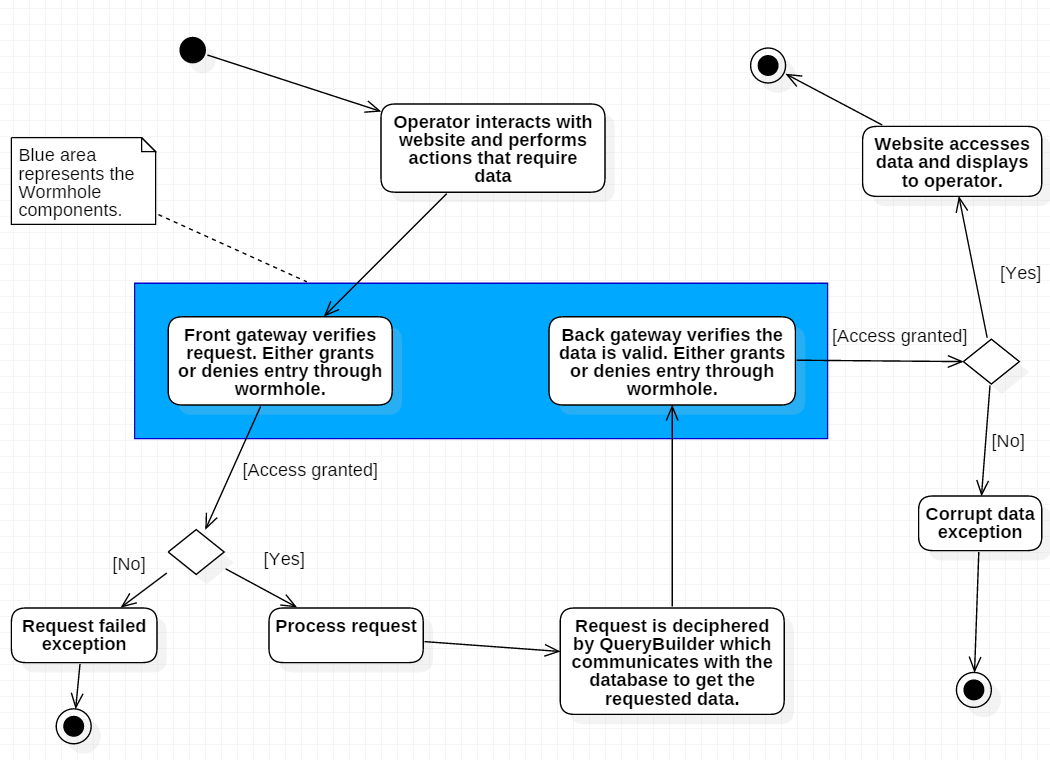
Choosing the “Registered Accounts” option queries each registered account that is stored within the database. This section of the website was built merely with the intention to display that our team is able to pull and display information previously stored.

Clicking the “My Account” text brings up a log in page. We can either log in from here if we already have an account created with an email and password or under the blue “Log in” button we can register if we are new to the site. Say we were to be a new user, another registration component would load allowing the user to enter their email address and a password twice (they must match to move on). After completing that a verification link would be sent to that email to check that it was legitimate.

**High Level Design**



Herein lies the general flow of communication throughout the website. The operator will interact with the **Javascript Front-End**, which consists of many methods for each part of website. The three arrows demonstrate that the **Javascript Front-End** can communicate with **the Front-Back Interpreter** in many different ways. This includes methods for transactions, displaying tickets and registering accounts on the website. The **Front-Back Interpreter**, which consists of both Javascript and Python code. The Front-Back Interpreter is the central interface that will be used when calling methods within the Python code. The calls will be passed to Python in this interface and the python will return its data to the front end through this interface upon calls that make requests for data. The single IN and OUT arrows show how communication components can only occur through one path. The **Python Back-End** is responsible with handling all data related activities, such as communicating with the Database and performing operations that the user requests through the **Javascript Front-End**. The three arrows show how the **Python Back-End** can communicate with the **Database** through more than one path, as in many Python methods can access or update the **Database**.

**Wormhole**

Wormhole is a group of components tasked with creating centralized communication between the Javascript front end and the Python back end of the software. The front gateway, written in Javascript, performs calls to the Python methods. The front gateway is the only component that will make calls to the Python methods. All other Javascript components shall make their requests to the front gateway interface. A Javascript exception handler is another component of the front gateway, which will verify the requests are valid.

The back gateway is written in Python code and consists of methods that the Javascript code can use to retrieve the data requested. Another exception handler, written in Python, will be part of the back gateway. This exception handler will ensure that the data received from the database is valid and decide if it is capable of entering the wormhole back to the operator on the website.

**Use Cases**

1. Seller use cases
   1. Seller creates account
   2. Seller modifies account information
   3. Seller deactivates account
   4. Seller inputs payment information in case of non-working tickets
   5. Seller views past transactions (sales)
   6. Seller views expired ticket listings
   7. Seller views deactivated ticket listings
   8. Seller starts listing of tickets
      1. Provides quantity of tickets
      2. Provides section, row, seat number
      3. Provides quantities that people can buy at a time (multiples of 2, all tickets together)
      4. Provides disclosures on tickets (obstructed view, no alcohol section, wheelchair accessible, etc)
      5. Provides comments on tickets (Early entry access, Aisle seats, concession credit, etc)
      6. Sets pricing for tickets (all tickets priced the same amount)
      7. Uploads PDF files
   9. Seller modifies listing of tickets
      1. Deactivates listing
      2. Change price of listing
      3. Change quantities of tickets that can be purchased in listing
2. Buyer use cases
   1. Buyer Account Functions
      1. Buyer creates account
      2. Buyer modifies account
      3. Buyer deactivates account
      4. Buyer inputs payment information
      5. Buyer views past transactions
   2. Searching functions
      1. Buyer searches for a specific game
      2. Buyer searches for games a specific team is playing in
      3. Buyer searches for best value (price)
      4. Buyer wants to browse
   3. Buying functions (Once on event listing page)
      1. Buyer filters tickets by quantity available
      2. Buyer filters tickets by aisle seats only
      3. Buyer filters tickets to exclude obstructed view seats
      4. Buyer filters tickets by handicap accessible only
      5. Buyer buys subset of tickets in a group
      6. Buyer buys all tickets in a group
      7. Buyer buys tickets, does not have payment info on file, needs to input at time of sale

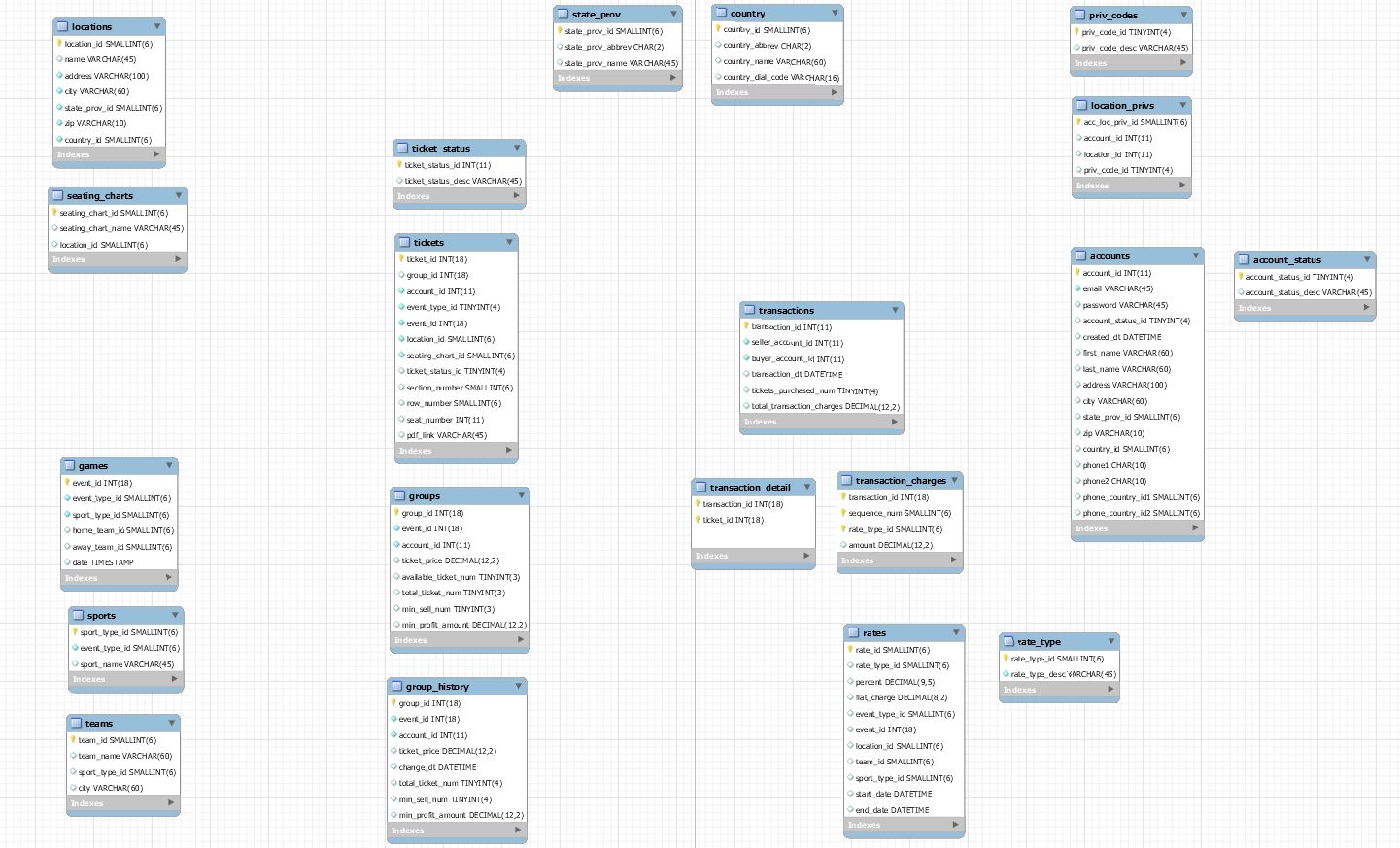
**Webpages/Navigation**

1. Home
   * Front page of website with large image of Wells Fargo Center
   * Top navbar for navigation to other pages
   * Scroll down for a calendar view of upcoming games
     + The list of events on the calendar is drawn from the /events endpoint
     + Selecting an event on calendar redirects you to a page for purchasing tickets for event
2. Buy
   * Calendar view
     + Show all upcoming games on a calendar
     + The list of events on the calendar is drawn from the /events endpoint
     + Selecting an event on calendar redirects you to a page for purchasing tickets for individual event
   * Individual event page
     + Shows seating chart of arena with available tickets in a panel on the right
       - Ticket data pulled from the /tickets endpoint
     + Individual sections are selectable.
       - Mouseover of a section shows view from that section
         * Image pulled from /view endpoint
       - 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
       - Asks for payment info
       - Upon transaction completion, tickets transferred via email
3. Sell
   * Product input page
     + Require user login
     + Collect all relevant ticket information from seller
       - Event
       - Event date
       - Section, row, seat number(s)
       - Price
       - Seller credit card information in case ticket is fake
     + Once information filled out, data sent to /sell endpoint for saving/verification
4. My Account
   * If not logged in, redirects to Login page.
   * If logged in, you can view account info
     + Update password, email address
     + View purchase history
     + View sale history
       - Cancel item currently being sold
5. Login
   * Login with email, password
   * Uses /login endpoint for credential authentication
6. Registration
   * Create account with email, password
   * Uses /register endpoint to send credentials to database
   * Backend sends authentication email to provided email address
   * Clicking link sends user to our webpage 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

Python Microservices/Endpoints

(Note: all input and output in JSON format)

1. 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.
2. 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.
3. 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.
4. Update Account (/update-account)
   * Input: Updated account information
   * Output: If successful, account information updated in accounts table. Outcome of account update attempt returned to front end.
   * General algorithm: Update account entry in accounts table with new information. The success/failure of the update is returned to the front end.
5. Account History (/account-history)
   * Input: History type (purchases or contributions), account id
   * Output: List of tickets purchases or sold returned to front end
   * General algorithm: Query database for all transactions associated with account id, return information to front end.
6. Events (/events)
   * Input: Date range, Location (optional), Team (optional)
   * Output: List of events that fit provided criteria
   * General algorithm: Query games table for games that have a date within date range, optional location match, and optional home/away team match. Return all query results to front end.
7. Tickets (/tickets)
   * Input: Event, Section(s)
   * Output: List of tickets for section(s)
   * General algorithm: Query tickets table for tickets which are available for purchase, match the event, and optionally for the section(s) provided.
8. Views (/view)
   * Input: Location, Section
   * Output: Image of view from section sent to front end
   * General algorithm: POST request to Amazon S3 to get picture for location/section combination, return image url to front end for display
9. Contributions (/contributions)
   * Input: Seller account email, location, event, section/row/seat (for each ticket), price (for each ticket), pdf of each ticket
   * Output: Contribution saved in database, outcome returned to front end
   * General algorithm: First we will determine if the section/row/seat combination is valid for the location. If valid combination, query tickets table to see if identical tickets are already for sale. If no match, insert new record for new ticket contribution associated with the seller. Save ticket pdf.
10. Purchases (/buy)
    * Input: Ticket ID(s), buyer account email
    * Output: If successful, transaction information added to database. Tickets table updated to indicate tickets sold. Outcome returned to front end.
    * General algorithm: Query /tickets table to determine if items in question are available for sale. If they are, change status to sold and add transaction to transactions table. Complete transaction between buyer and seller. Email buyer pdf of tickets.

Database Specification

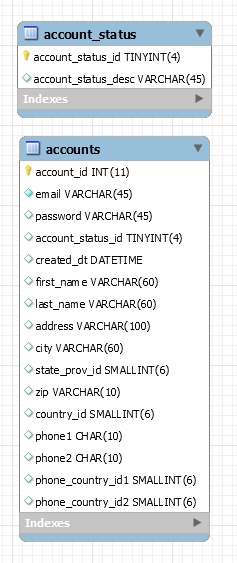
****

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).

Figure - Account-related tables

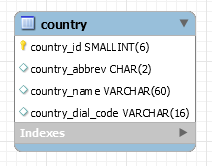
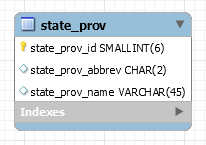
Accounts contain detailed contact information including email, first name, last name, and address information. Notice that state\_prov\_id and country\_id are primary keys referencing unique state and country values for the account in tables state\_prov and country.

Figure - Country table

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:

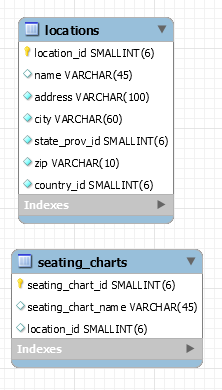
The country table stores country values with a column for dial codes for those countries.

Figure - state\_prov table

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.

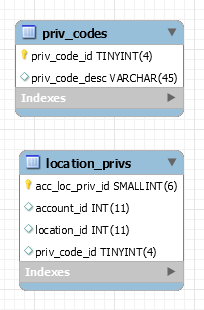
Locations have seating charts. A seating chart is a configuration of seating arrangements made for a specific one-time or reoccurring event. A location can, and must have many different seating charts. Examples of seating charts include the Philadelphia 76ers seating configuration in the Wells Fargo Center, or the seating configuration for a Flyers game in the Wells Fargo Center. This is a good example where one location (the Wells Fargo Center) has multiple seating charts for different events (Flyers and Sixers).

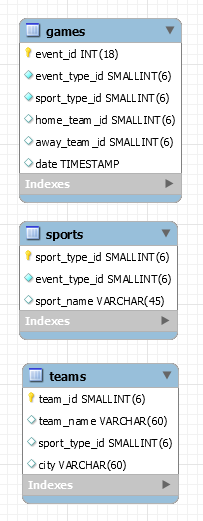
Figure 5 - Location-related tables

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.

Figure 6 - priv-related 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.

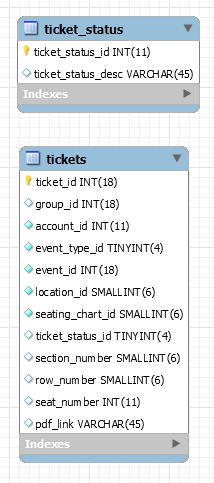
Games, Sports, and Teams:

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.

Tickets:

Figure 7 - Sport-related entities

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.

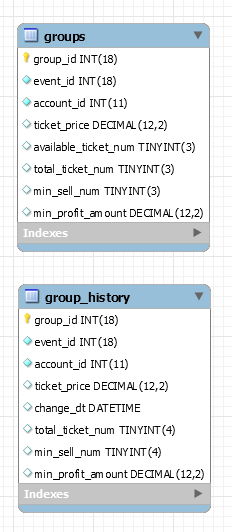
The ticket\_status table stores different statuses that can exist for tickets: ‘available’, ‘sold’, ‘cancelled’, etc. This status will change when tickets are uploaded, completed in a transaction, or cancelled.

The tickets table column pdf\_link will contain a relative path to the pdf file for the ticket. The account\_id for a ticket refers to the account that uploaded the ticket.

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.

Figure 8 - ticket-related tables

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.

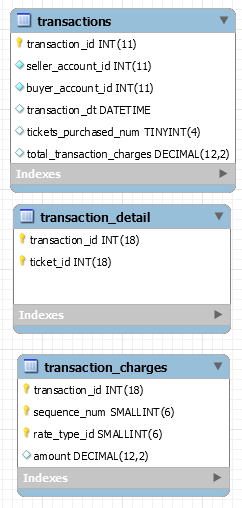
The group\_history table stores “snapshots” of the state of groups. When a group is originally uploaded, a row will be inserted in the group\_history with all of the same values as its corresponding row in the groups table, plus a timestamp column. Whenever a seller changes the information of one of 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.

Figure 9 - Group tables

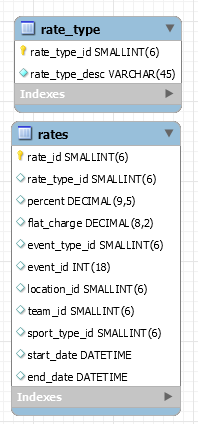
This table will allow pricing trends to be tracked, and other statistics to be collected about how listings are being updated.

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.

Figure 10 - Transaction tables

Rates:

The rates table stores several rows with certain attributes that are compared against the attributes of tickets during a transaction. If an attribute 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:



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.

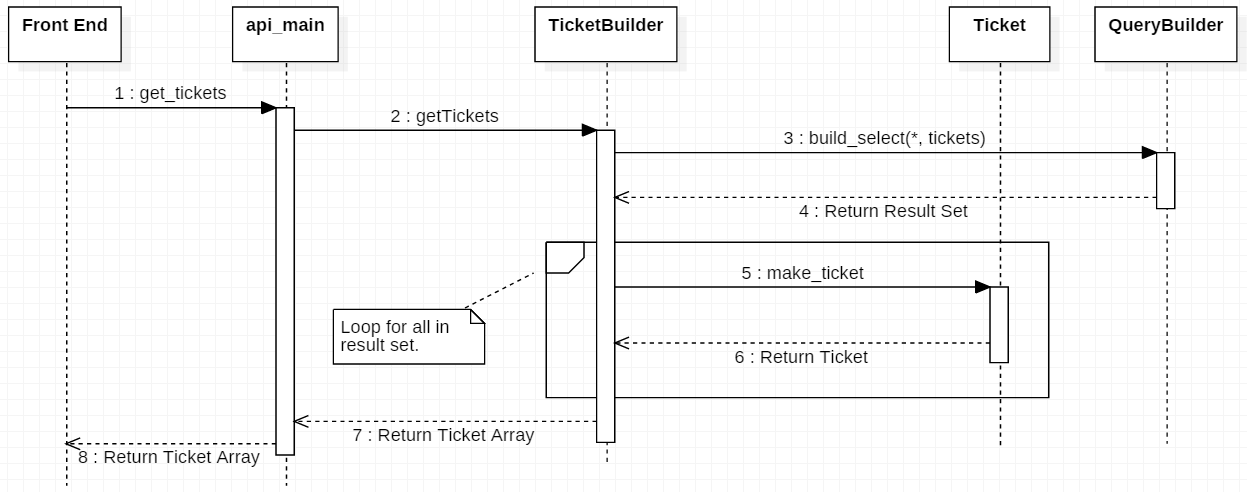
Figure 11 - Rates tables

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.

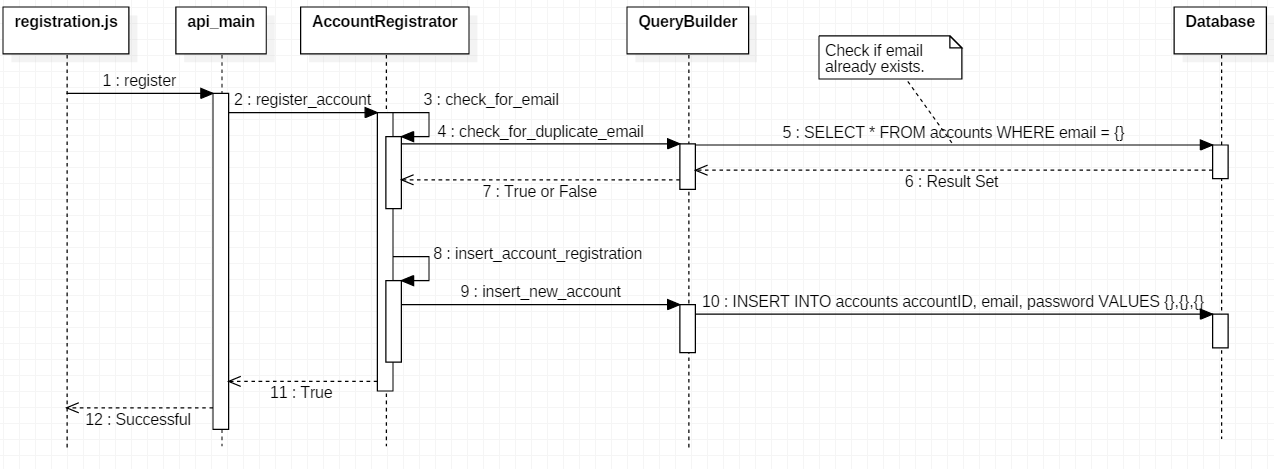
Conclusion:

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.

**Diagrams**



The sequence diagram above shows the flow of operation from each component when retrieving a set of tickets to display in the front end. The Javascript Front-End will access methods in api\_main, a part of the Front-Back Interpreter, which will get the tickets from the TicketBuilder. The TicketBuilder will query for the tickets based on the search filters specified by the Operator on the front end. The result will be fetched from the database by the QueryBuilder, who is responsible for accessing and retrieving data from the database. The result set is returned to TicketBuilder, where a loop will make a Ticket object from each row of data returned in the Result Set. The Tickets will be packaged into an array and returned back up to the front end where it will be display to the operator.



The sequence diagram above depicts the operator action of registering an account successfully on the website. The registration Javascript file will call the register method in api\_main. The api\_main class will then call the register\_account method in AccountRegistrator, where it will check for duplicate emails. To check for duplicates the AccountRegistrator attempts to SELECT the email inputted by the operator from the database through the QueryBuilder. If this is successful, then the email exists otherwise the email does not exist yet and execution can continue. The valid email is then inserted into the database with information. Successful insertion of the new email is replied back to the operator.

**Implementation Goals by Mid-Assessment:**

1. Display ticket data from the database on event listing page
   1. Add functionality to display all ticket listings for a specific event on the side panel of the seating chart. This involves a call to the database in order to retrieve this information.
   2. Curtis will work on this
2. Add functionality to perform a basic transaction
   1. This would involve allowing a user to select a listing of tickets, choose to buy them, and allow them to input payment information and purchase the ticket listing.
   2. Tom will work on this
3. Add an account registration form
   1. Add functionality so that a user can input their name, address, email, and password and create a user.
   2. Derek will work on this
4. Add a basic filter system on a listing
   1. Add functionality so that when a user clicks on a section, it filters the tickets shown to be only the tickets from that given section.
   2. Jon and Chris will work on this
5. Backend Architecture
   1. Work on completing Wormhole
   2. Data processing in backend with Python
   3. Exception Handling
   4. Anthony will work on this

**Exclusions from Final Submission:**

Running list of project limitations (exclusions)

1. Only doing basketball, sixers, home games, (at wells fargo center)

2. We are doing "simulated" transactions (no funds are being transfered)

3. Only able to create ticket listings for tickets in the same row (same section as well)

4. Not doing disclosures. (We only doing extras [early entry, aisle seat, handicapped accessible])

5. 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.