TrustTheTickets.com  
Detailed Design Document

Initial Draft Date: October 9th, 2017

Created by:

Anthony Orio

Christopher McKane

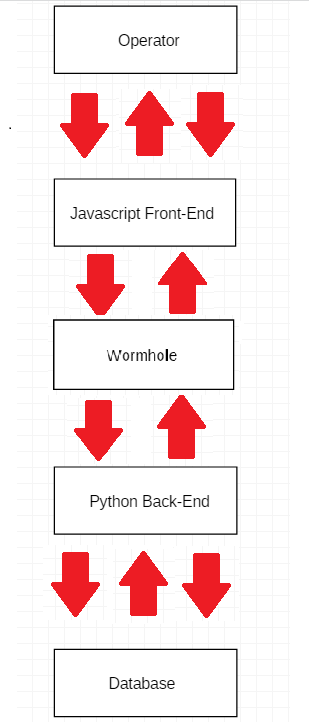
Curtis Baillie

Derek Gaffney  
Jon D’Alonzo

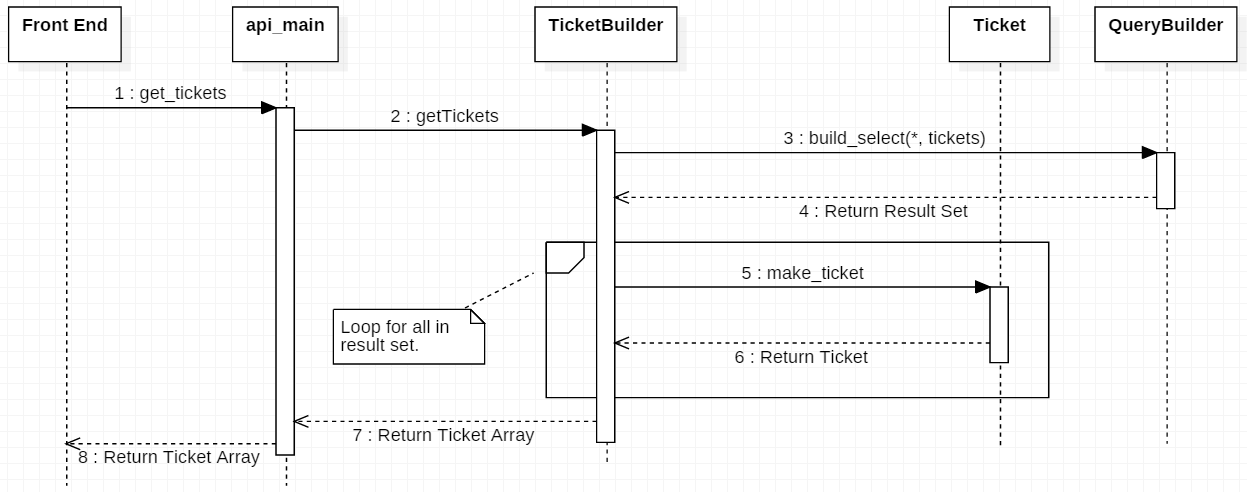
Thomas Harker

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

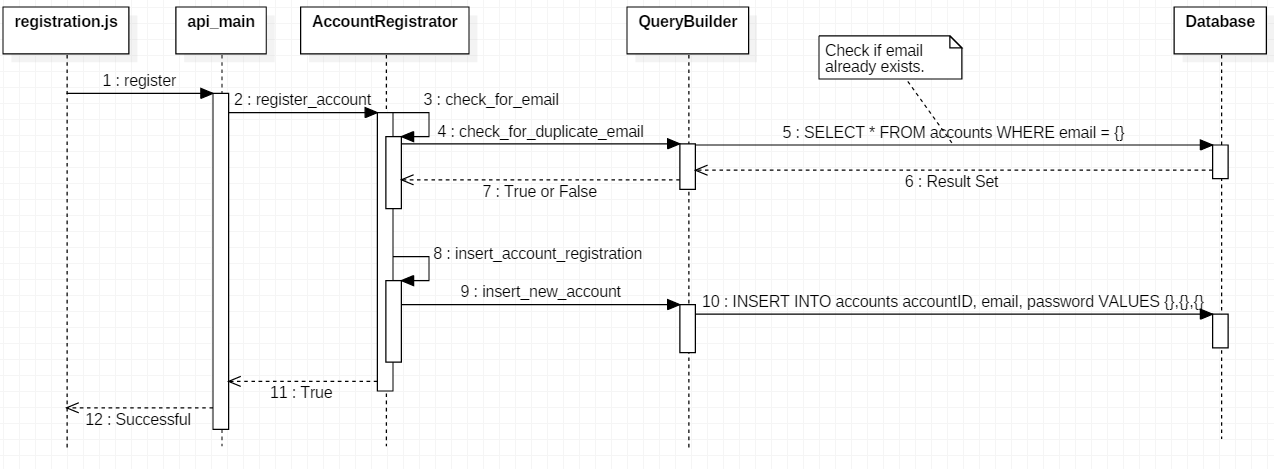
**High Level Design**



Here 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**.



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.

**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.

**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

1. 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

**Implementation Goals by Mid-Assessment:**

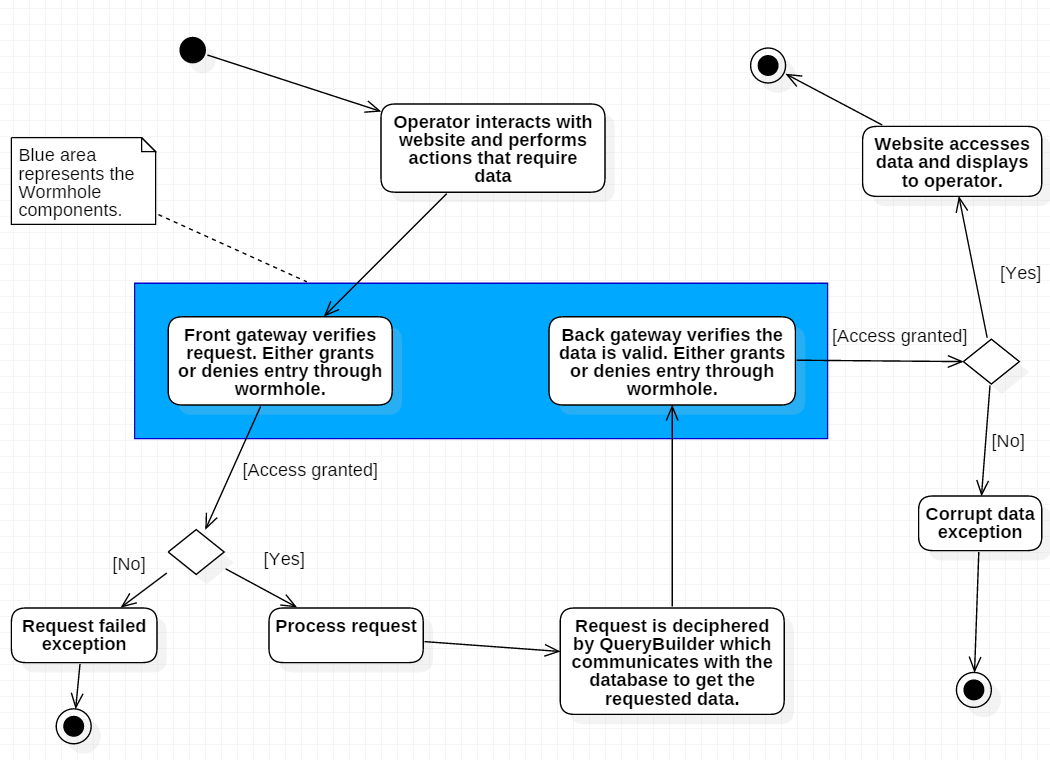
1. Display ticket data from the db on event listing page
   1. Add functionality to display all ticket listings for a specific event on the side panel of the seating chart. This involves a call to the db in order to retrieve this information.
   2. Curtis will work on this
2. Add functionality to perform 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

General Use Case

Handling box seats vs. section seats

System use cases

**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.