Lafayette College Senior Project 2020
Developer Manual

Cameron Zurmuhl **Joseph Teddick** **Matthew Gerber** **Xingwen Wei**

Prior Developers

Cameron Zurmuhl (cameron.zurmuhl@verizon.net)
Joseph Teddick (josephteddick@gmail.com)
Matt Gerber (gerberm@lafayette.edu)
Xingwen Wei (xingwen2018@gmail.com)

Getting Started

The public repository for the project is hosted on GitHub:
https://github.com/20zurmca/SeniorProject

The README.md file contains a brief description of the directory structure of the repository.

We recommend installing Anaconda as a package manager for Python. You can create virtual environments through Anaconda. You can also use pip if you so choose. Our server has both installed. You can find our installation of Anaconda in the /opt/anaconda directory.

The python dependencies for our repository are:
Django=3.0.3
psycopg2=2.84
python-dotenv=0.13.0 (install with pip)

And these dependencies should be installed into your virtual environment. You can create a virtual environment with either pip or anaconda.

To run the application locally, you need to use ssh tunneling to connect to our Postgres database on server 139.147.9.235 through port 3333 on your local machine. Notice Postgres is running on port 5432 on our server.

```
ssh -L 3333:127.0.0.1:5432 zurmuhlc@139.147.9.235 -N
```

Instead of zurmuhlc, use your username. You will notice that the connection will hang after you enter your password. Do not be alarmed about this; it is working. If you want to change the port number, amend settings.py int ./soccersite/soccersite, line 87:

```
DATABASES = {
   'default': {
      'ENGINE': 'django.db.backends.postgresql_psycopg2',
      'NAME': 'soccer',
      'USER': 'socceruser',
      'PASSWORD': 'seniorproject',
      'HOST': 'localhost',
      'PORT': '3333',
   }
}
```

Notice that our psql database name is ‘soccer’.

To launch the application from your localhost, open up the anaconda command prompt (or any other terminal with Python if you are using pip only), and activate your virtual environment where you installed the dependencies. Then change directories to you are in the SeniorProject/soccersite directory. Before launching the app, make sure DEBUG=True in /soccersite/soccersite/settings.py (line 31) so Django hosts your static files. Launch the app with command `python manage.py runserver.` Make sure DEBUG=False before redeploying to production.

Front End

The user facing side of our web application utilizes HTML 5, CSS 3, and JavaScript. The main page of the application contains the drop down menus, the map, and the data table. Additionally, we make use of the Google Maps Api, bootstrap 4, tail.select, jquery, and ajax. Using the toolbar at the top of each page, the user can access the about page, scraped data and manual data upload pages, and the restore data page, as well as the built in Django authentication. Each page has a corresponding HTML and CSS file. Below is a list of the JavaScript files with descriptions of the most important or confusing methods. For more details on the implementation, refer to the commented code.

- map.js - Contains implementation for google map api, data table
    - loadData(map, playerData)
        - Initializes all import data, creates heat map, markers, and info windows, and sets up calculator method for clusters
    - HeatMapControl(controlDiv, map)
        - Adds the functionality to toggle on and off the heatmap
    - ZoomControl(controlDiv, map)
        - Adds the functionality to auto zoom on the currently displayed markers. This works by extending the bounds of the map and calling the fit bounds method of the Google Maps API (described below)
    - MarkerControl(controlDiv, map)
        - Adds the functionality to toggle on and off the markers. This method contains the same implementation for cluster calculator method
    - initMap()
        - Initializes instance variables of Google map and sets up divs for the heat map, zoom and marker control buttons
- selectors.js - Contains a dictionary for the college leagues and sets placeholder values for all drop down menus
    - leagueSelector.on("change", function()
        - Based on the dictionary of leagues, selects and deselects colleges respectively based on the league chosen
- form.js
    - Script that handles the AJAX calls with the submit button.
        - getCookie(c_name) is used to get the crsf_token for a post request
        - _findAssociativeIndices(roster_year) is used to help obtain the most recent roster-year information for a player

        - _getCurrentDataElement(data, association) returns the most recent element for the latest possible roster year that has a corresponding entry.
        - convertToInches(height) converts a height like 6'0 to inches. This is why when scraping, it is important not to include the " character at the end.
        - The ajax call on submit sends the selected drop down data to the view and upon success, calls loadData(map, response['players']) in map.js.
            - An initial datatable is created in the frontend
- homePageButton.js - Contains implementations for all the buttons on the home page
    - submit()
        - Resets various attributes of the map and displays the loader
    - parseCell()
        - A helper function used in the csv method
    - csv()
        - Loops through the data currently in the table and (using the parse cell method) exports them to a csv file called "SoccerQuery.csv"
- aboutPageButtons.js - Contains buttons to go to the LinkedIn pages of the developers

Below is a list of APIs and frameworks used in the front end of the application as well as the common methods associated with them:

- Google Maps Api - https://developers.google.com/maps/documentation/javascript/tutorial
    - Google Maps Map
        - The map is initialized in the initMap function. This sets the initial bounds, initial zoom, and various control options. For further documentation, see the link above
        - The Google API Key is restricted for *.lafayette.edu/* websites
    - Map.fitbounds(bounds)
        - Using the newly assigned map boundaries from the call to bounds.extend, zooms the map to fit
    - MarkerCluster.setCalculator(function(markers, numStyles))
        - Creates a custom calculator method to change how the marker clusters are grouped. For our implementation, we wanted the pins to cluster based on student count and not simply the number of pins. This method uses the student count at each marker and assigns an index based on the total number of students compared to the total number of marker pins. This calculator method returns the count and the index for each marker cluster
    - Google Maps Infowindow

        - A single infowindow instance is created when the data is initially loaded in. When a pin is clicked on, both the location and text on the infowindow are updated based on the school
    - Google Maps Visualization Heat Map Layer
        - Built in function to create the heat map. The map of the layer is set to the instance of the main map. The data is pushed to the heat map in loadData method
- Bootstrap 4 - https://www.w3schools.com/bootstrap4/bootstrap_get_started.asp
    - Our application uses bootstrap 4 as the default style guide for all the pages. This is set up in all html files
- Tail.select - https://www.npmjs.com/package/tail.select
    - Our application uses tail select in order to improve the functionality of the drop down selectors on the main screen. It is used in the creation of the selectors in selectors.js
- Ajax - https://api.jquery.com/jquery.ajax/
    - In combination with jQuery, our application uses ajax to make requests for data from the server. This can be seen in forms.js. Given the values selected by the user, an ajax post request gets the matching data from the server

Below are links to the pictures used on the site. According to their license we had permission to use them.

Main photo:
https://pixnio.com/media/corner-lawn-soccer-ball-sport-football

Footer photo:
https://www.needpix.com/photo/download/544457/corner-sports-field-soccer-football-stadion-grass-green-lawn

Back End

Scraping the Player Profiles & Data Collection

The data provided for the project comes from multiple areas of the web. Scraping was completed for 2006-2018, but future years need to be scraped, starting with 2019. First, the roster data comes from college's roster sites. Here is an example of a roster site to scrape from: https://goleopards.com/sports/mens-soccer/roster

Your task is to write/use code to scrape the HTML to gather the necessary attributes from each of the players and dump the output into a csv file. Here are the attributes to focus on:

Id (a unique integer identifier for the row. This is to satisfy a Django requirement for having a primary key. Does not have foreign key references to other tables. You don't have to actually put in an id, you just need the column name. Django will automatically update the id for you.)
Roster Year
Player Number
First Name (make sure the first name matches across files)
Last Name (make sure the first name matches across files)
Year (as in class year, i.e Sophomore, etc.)
Position1
Position2
Position3
Height
Weight
Home Town
State or Country (if state make sure the abbreviation is 2 letter uppercase. If country, make sure it is spelled out fully)
High School (if this field is null, set it equal to the alternative school)
Alternative School
College
College League
Bio Link

The Roster Year is a Fall year of the season, so it is not a range like 2019-2020. It would just be 2019. Make sure that the year is spelled out fully, like Freshman, Sophomore, Junior, Senior, Graduate, Redshirt, etc. Although up to three positions could be given, we use only position1 in our application, as per Coach Bohn's recommendation. But do include all three in the file, even

if two are null. Make sure the height does not have the double-quote character at the end of the string. Formatting should be like: 5'6 not 5'6". Make sure states are abbreviated to two-capital letters. For a dictionary mapping of the colleges to the college leagues which can be used for reference, see SeniorProject\scripts\SoccerScrape/LeagueDictionary.py.

We recommend using the scrapy library for scraping, which is what the scripts in the SoccerScrape directory uses. You may choose to use some or all of the code in that directory, but be aware that some code may have to change since HTML changes by the year. So some websites may require custom code. To see an example of an output csv file, look in the SeniorProject/data folder. You'll see roster_data.csv, as well as starter_data.csv and accolades.csv

Next, you will have to scrape games played-games start data to determine if a player is a starter. A player is a starter if he played >=50% of possible starts. Possible starts are the number of games the teams play. An example site for this data can be found here: https://goleopards.com/sports/mens-soccer/stats

Make sure to scrape the "Individual" "Overall" "Offensive" and "Goalkeeping" tabs.
Keep in mind that the format that you see on a particular site may be consistent or inconsistent among other sites. Here are the attributes to scrape:

Id (a unique integer identifier for the row. This is to satisfy a Django requirement for having a primary key. Does not have foreign key references to other tables. You don't have to actually put in an id, you just need the column name. Django will automatically update the id for you.)
Roster Year
Number
First Name (make sure the first name matches across files)
Last Name (make sure the last name matches across files)
Potential Starts
GP (Games Played)
GS (Games Started)
Is Starter (must calculate. GS/Potential Starts >= 50%)
College

Last, you will have to scrape conference award pdf files to determine which players got an all-conference award. You can find these pdf files on the websites for the conferences. Try to look for Men's Soccer > Record Book. Here is an example: https://patriotleague.org/documents/2019/8/27/2019MSOCRecordBook.pdf

See Coach Bohn for more guidance on where to find the data.

Here are the attributes to scrape from the pdf files:
Id (a unique integer identifier for the row. This is to satisfy a Django requirement for having a primary key. Does not have foreign key references to other tables. You don't have to actually put in an id, you just need the column name. Django will automatically update the id for you.)
Roster Year
First Name (Make sure first name matches across files)
Last Name (Make sure last name matches across files)
Accolade
College

The SeniorProject/scripts directory already contains some python scripts to scrape data. Again, some sites may have changed from when the code was written, so you will have to make a decision to either start from scratch or amend the codebase.

I will now describe the files in the SeniorProject/scripts directory

The three python files at the top level are for geocoding school names to latitude and/or longitude. You probably won't need to use these, as we already compiled a database of high school data to match schools against. If you need to use them, be aware that boardingschoolgeocoder.py and googlegeocoder.py need Google API keys for geocoding.

SeniorProject/scripts/SoccerScrape/CurrentRosterYear.py is supposed to parse the current roster year out of a string like 2019-20, since this was a common case I came across. I think this parser may be broken for some sites.

SeniorProject/scripts/SoccerScrape/items.py define scrapy items to encapsulate the scraped data. The two items are Playersand Starter. See scrapy documentation for more information about items.

SeniorProject/scripts/SoccerScrape/LeagueDictionary.py contains a dictionary that maps the conferences (or leagues) in question to a list of their colleges in question. The get_college_from_url(urlDomain) gets a college from a base url. check_league(urlDomain) returns a league from a base url.

Middlewares.py came from scrapy and is unchanged.

Pipelines.py defines how to process scraped items. Scraped items go through the pipeline. The actual writing to csv files happens in the pipelines. You will most likely have to redefine some of the logic for future years since this logic was tailored for 2006-2018 data.

Player.py defines a class for representing a Player. Uncertain if this class was in use since bioLink is missing from the class.

The SeniorProject/scripts/SoccerScrape/spiders directory contains the scripts that do the web scraping. Schools with unique HTML get their own script. Whatever schools have similar HTML to another are grouped. For example, TableSpider.py has common site formats that are in table layout. TableSpiderHarvard.py is unique to Harvard.

.SeniorProject/scripts/SoccerScrape/data/append.py. This script was designed to transfer data from text files to a .csv file. It shouldn't be needed anymore.

(Important) .SeniorProject/SocerScrape/data/dataclean.py finds potential name matches among roster data paired with accolade data and roster data paired with starter data. Future starter data names should match with roster data names because those data come from the same source (a website). Accolade data comes from pdf files. So using fuzzy string matching, match_names_roster_conference() will find potential matches. It is up to the user to confirm matches and change any discrepancies. match_names_roster_starter() shouldn't be needed since going from 2019 onward, all roster data and starter data is organized in a cohesive website. In years like 2006, schools kept data in loose files still. Duke may be an exception to this rule.

.SeniorProject/SocerScrape/data/DataCollector.py combines all the scripts into one. This script also contains the scripts to scrape the pdf files. The logic may need to be changed for future pdf files. You could also put the pdf data in text files as you can observe in the conference_data folder. You can update the pdf files in the conference_data directory.

.SeniorProject/SoccerScrape/data/Starter.py defines class logic for a Starter.

Our best advice is to see what still works, see what is understandable, and see what is expendable. Perhaps you'd want to start from scratch because of educational reasons or the codebase just isn't relevant anymore for the current task at hand, which is to get the most current data, not go back up to 14 years and get all the data. Your problem will certainly be easier and will require less code. Just make sure the output is three .csv files that look exactly like roster_data.csv, starter_data.csv, and accolades.csv as seen in SeniorProject\data. Make sure player names match across the files you scrape.

Django Application Backend

The backend to our application is in views.py. The file is sufficiently commented and contains the logic for filtering, uploading, and restoring data.

Database

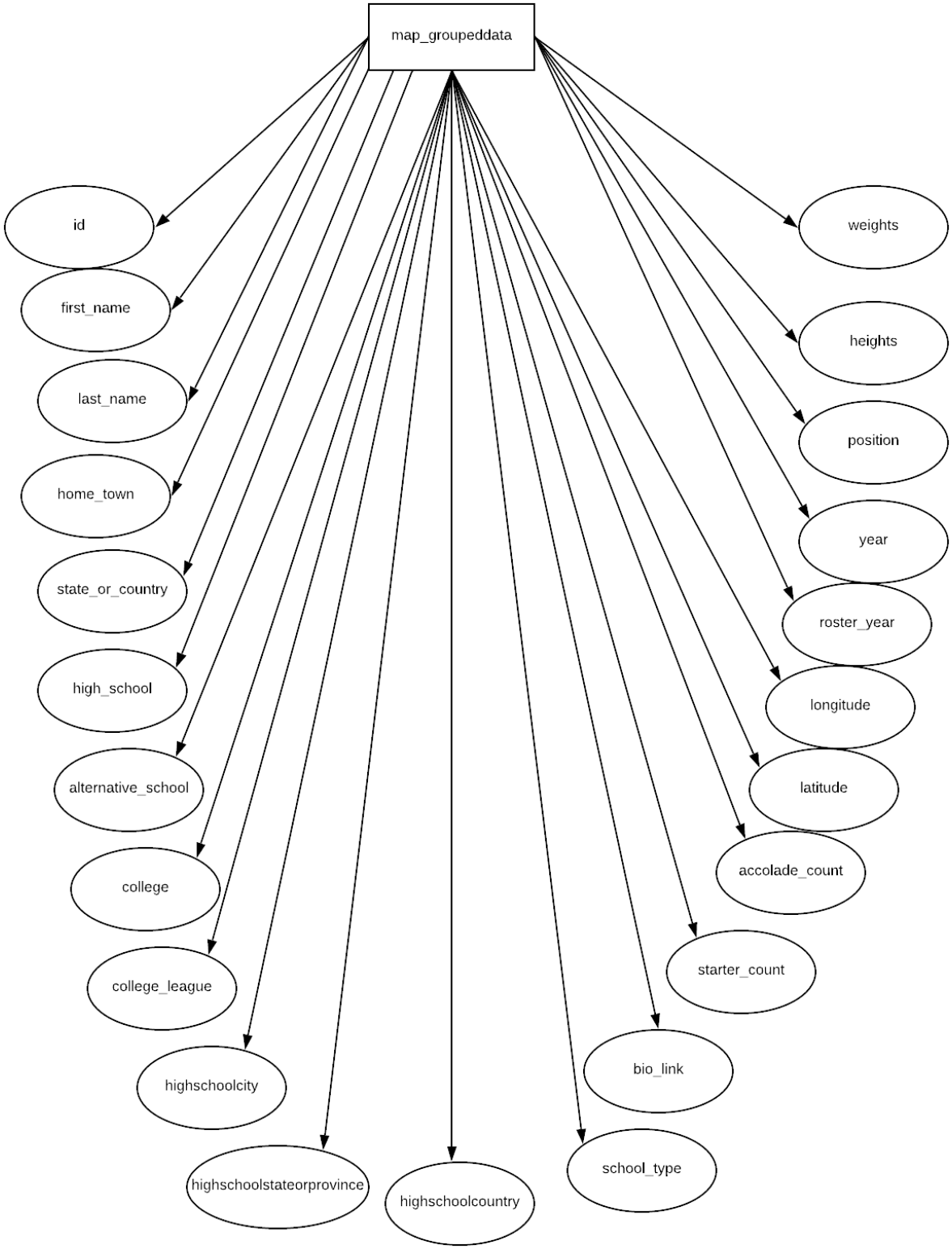
The goal of this section is to understand how the database is organized, what logic is used to match High Schools, and how to maintain the life of the database.

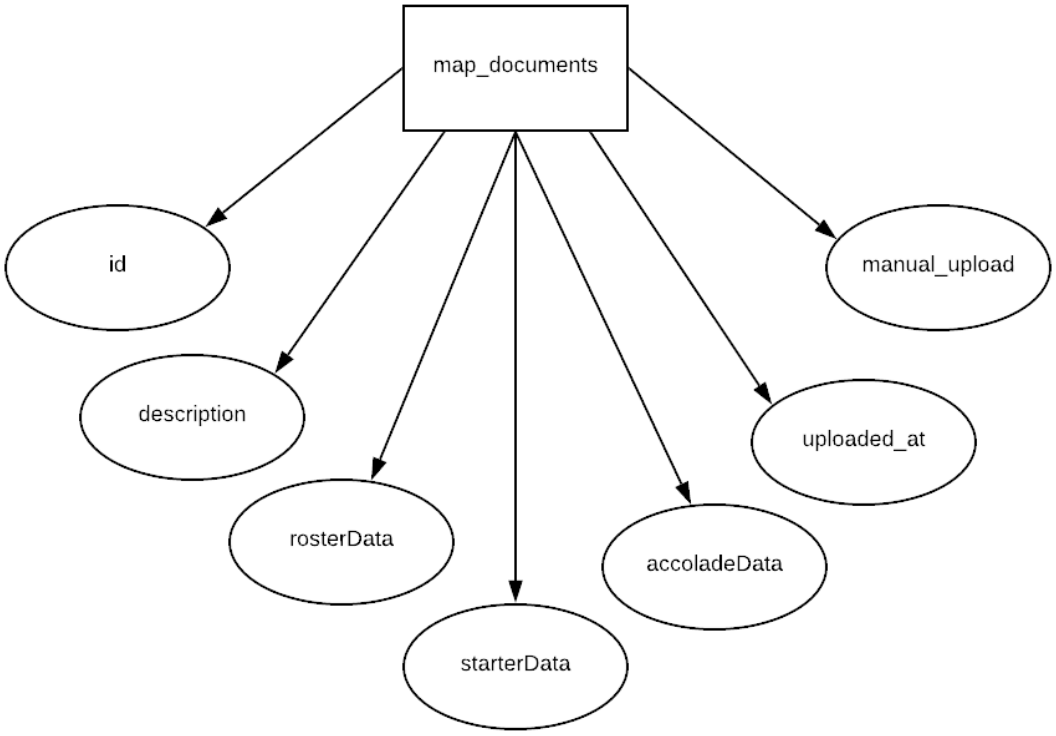
Data Organization (Table Definitions and ER Diagram)

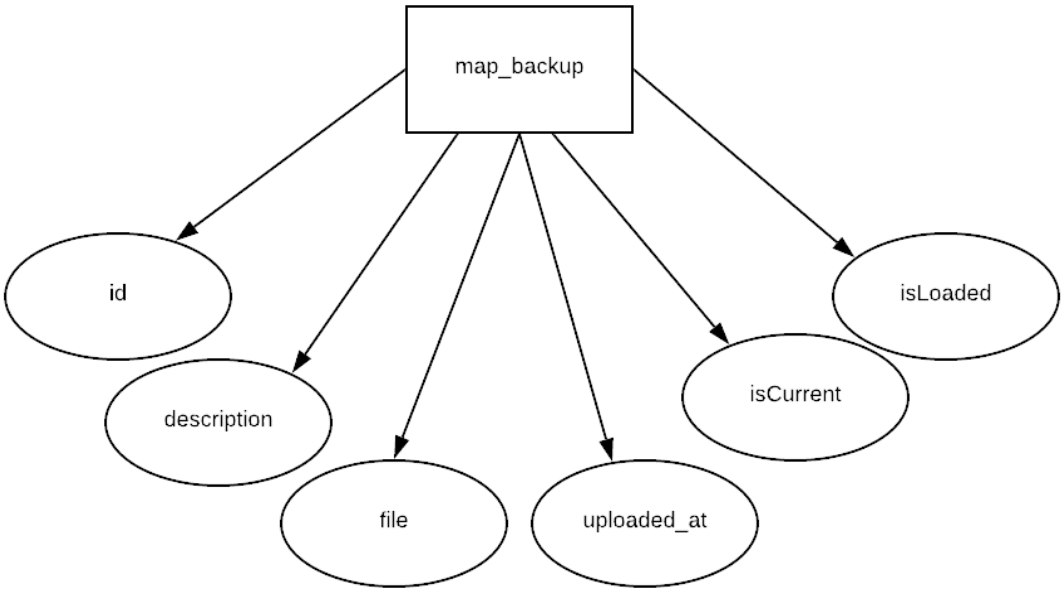
Automatically generated tables by django:
auth_group, auth_group_permissions, auth_permission, auth_user, auth_user_groups, auth_user_user_permissions, django_admin_log, django_content_type, django_migrations, django_session

Tables created by us:
map_accoladedata - stores the original scraped accolade data (awards they received)
map_backup - stores the information and file of a backup of the database
map_documents - stores the scraped files to be uploaded and some general information
map_groupeddata - stores the grouped up, matched player information
map_highschooldata - stores all the information about the high schools
map_highschoolmatchmaster - stores the matched player information
map_rosterdata - stores the original scraped roster data
map_starterdata - stores the original scraped starter player data
unique_boarding_schools - stores the unique boarding schools from the master high school table

Our database doesn't really have standard entity relationships as it is simply a database containing the data that we want to display on the front end website. However, we created this diagram seen below to display the three tables that are actively used by the front end. The map_groupeddata table is the grouped up data that contains all the information about each soccer player. The map_documents table contains the documents that are being uploaded by the 'Upload Scraped Data' page and has a trigger on insert that will join new data and merge it with the existing data to update the map_groupeddata table. Lastly, the map_backups table contains the backed up files of the database. This is used for the version control functionality and is added to each time a backup is generated (on scraped data upload). Once again the diagram of these tables is seen below.


map_groupeddata
id
first_name
last_name
home_town
state_or_country
high_school
alternative_school
college
college_league
highschoolcity
highschoolstateorprovince
highschoolcountry
school_type
bio_link
starter_count
accolade_count
latitude
longitude
roster_year
year
position
heights
weights


map_documents
id
description
rosterData
starterData
accoladeData
uploaded_at
manual_upload


map_backup
id
description
file
uploaded_at
isCurrent
isLoaded

Data Dictionary

Table Name	Attribute	Description	Data Type
map_accoladedata	id	Unique id of the player	integer
map_accoladedata	roster_year	Year the player is on the roster	integer
map_accoladedata	first_name	First name of the player	character varying(50)
map_accoladedata	last_name	Last name of the player	character varying(50)
map_accoladedata	accolade	Accolade the player received that year	character varying(20)
map_accoladedata	college	College of the player	character varying(50)
map_backup	id	Unique id of the backup	integer
map_backup	description	Description of backup entered by user	character varying(255)
map_backup	file	Filename of backup	character varying(100)
map_backup	uploaded_at	Date and time of backup creation	timestamp with time zone
map_backup	isCurrent	Is the backup current?	boolean
map_backup	isLoaded	Is the backup loaded?	boolean
map_documents	id	Unique id of the document	integer
map_documents	description	Description of document	character varying(255)
map_documents	rosterData	Name of the roster	character

		file uploaded	varying(100)
map_documents	starterData	Name of the starter file uploaded	character varying(100)
map_documents	accoladeData	Name of the accolade file uploaded	character varying(100)
map_documents	uploaded_at	Date and time the files were uploaded at	timestamp with time zone
map_documents	manual_upload	Was the upload a manual uploaded?	boolean
map_groupeddata	id	Unique id of the player	integer
map_groupeddata	first_name	First name of the player	character varying(50)
map_groupeddata	last_name	Last name of the player	character varying(50)
map_groupeddata	home_town	Player's home town	character varying(30)
map_groupeddata	state_or_country	Player's state (if applicable) or country	character varying(20)
map_groupeddata	high_school	Player's high school	character varying(100)
map_groupeddata	alternative_school	Player's alternative school	character varying(50)
map_groupeddata	college	Player's college	character varying(50)
map_groupeddata	college_league	League of the player's college	character varying(50)
map_groupeddata	highschoolcity	City of the player's high school	text
map_groupeddata	highschoolstateorprovince	State (if applicable) or province of the player's high school	text
map_groupeddata	highschoolcountry	Country of the player's high school	text

map_groupeddata	school_type	Player's high school type	character varying(13)
map_groupeddata	starter_count	Years the player was a starter	integer
map_groupeddata	accolade_count	Number of accolades the player recieved	integer
map_groupeddata	latitude	Latitude of the player's high school	double precision
map_groupeddata	longitude	Longitude of the player's high school	double precision
map_groupeddata	roster_year	List of years the player was on the roster	integer[]
map_groupeddata	year	List of the years the player was on the roster	character varying(50)[]
map_groupeddata	position	List of positions the player held	character varying(50)[]
map_groupeddata	heights	List of heights the player recorded	character varying(5)[]
map_groupeddata	weights	List of weight the player recorded	integer[]
map_groupeddata	bio_link	Link to the bio link of the player	character varying(100)[]
map_highschooldata	id	Unique id of the high school	integer
map_highschooldata	city	City of the high school	character varying(50)
map_highschooldata	institution	Name of the high school	character varying(100)
map_highschooldata	stateorprovince	State (if applicable) or province of the high school	character varying(20)

map_highschooldata	country	Country of the high school	character varying(50)
map_highschooldata	latitude	Latitude of the high school	double precision
map_highschooldata	longitude	Longitude of the high school	double precision
map_highschooldata	school_type	Type of high school	character varying(20)
map_highschoolmatchmaster	id	Unique id of the player	integer
map_highschoolmatchmaster	roster_year	Year the player was on the roster	integer
map_highschoolmatchmaster	player_number	Player's number	character varying(10)
map_highschoolmatchmaster	first_name	Player's first name	character varying(50)
map_highschoolmatchmaster	last_name	Player's last name	character varying(50)
map_highschoolmatchmaster	year	Player's school year	character varying(10)
map_highschoolmatchmaster	position1	Player's primary position	character varying(20)
map_highschoolmatchmaster	height	Player's height in inches	character varying(10)
map_highschoolmatchmaster	weight	Player's weight in pounds	integer
map_highschoolmatchmaster	home_town	Player's home town	character varying(30)
map_highschoolmatchmaster	state_or_country	Player's state (if applicable) or country	character varying(20)
map_highschoolmatchmaster	high_school	Player's high school	character varying(100)
map_highschoolmatc	alternative_school	Players alternative	character varying(50)

hmaster		school	
map_highschoolmatchmaster	college	Player's college	character varying(50)
map_highschoolmatchmaster	college_league	League of the player's college	character varying(50)
map_highschoolmatchmaster	bio_link	Link to the player's college bio	character varying(100)
map_highschoolmatchmaster	is_starter	Was the player a starter that year?	character varying(1)
map_highschoolmatchmaster	accolade	Accolade the player received that year	character varying(20)
map_highschoolmatchmaster	city	City of the player's high school	character varying(30)
map_highschoolmatchmaster	institution	Name of the player's high school	character varying(30)
map_highschoolmatchmaster	stateorprovince	State (if applicable) or province of the player's high school	character varying(20)
map_highschoolmatchmaster	country	Country of the player's high school	character varying(50)
map_highschoolmatchmaster	latitude	Latitude of the player's high school	double precision
map_highschoolmatchmaster	longitude	Longitude of the player's high school	double precision
map_highschoolmatchmaster	school_type	Type of player's high school	character varying(20)
map_rosterdata	id	Unique id of the player	integer
map_rosterdata	roster_year	Year this player was on the roster	integer
map_rosterdata	player_number	Player's number	character varying(10)
map_rosterdata	first_name	Player's first name	character varying(50)

map_rosterdata	last_name	Player's last name	character varying(50)
map_rosterdata	year	Player's school year	character varying(10)
map_rosterdata	position1	Players primary position	character varying(20)
map_rosterdata	position2	Player's secondary position	character varying(20)
map_rosterdata	position3	Player's tertiary position	character varying(20)
map_rosterdata	height	Player's height in inches	character varying(10)
map_rosterdata	weight	Player's weight in pounds	integer
map_rosterdata	home_town	Player's home town	character varying(30)
map_rosterdata	state_or_country	Player's state (if applicable) or country of residence	character varying(20)
map_rosterdata	high_school	Name of the player's high school	character varying(100)
map_rosterdata	alternative_school	Name of the player's alternative school	character varying(50)
map_rosterdata	college	Name of the player's college	character varying(50)
map_rosterdata	college_league	League of the player's college	character varying(50)
map_rosterdata	bio_link	Link to the player's college bio	character varying(100)
map_starterdata	id	Unique id of the player	integer
map_starterdata	roster_year	Player's year on the roster	integer
map_starterdata	number	Player's number	integer

map_starterdata	first_name	Player's first name	character varying(50)
map_starterdata	last_name	Player's last name	character varying(50)
map_starterdata	potential_starts	Number of games that the player's team played that season	integer
map_starterdata	gp	Number of games the player played that season	integer
map_starterdata	gs	Number of games the player started that season	integer
map_starterdata	is_starter	Was the player a starter in the given year?	character varying(1)
map_starterdata	college	Player's college name	character varying(50)
unique_boarding_schools	city	City where the boarding school is located	character varying(50)
unique_boarding_schools	institution	Name of the boarding school	character varying(100)
unique_boarding_schools	stateorprovince	State (if applicable) or province of the boarding school	character varying(20)
unique_boarding_schools	country	Country of the boarding school	character varying(50)
unique_boarding_schools	latitude	Latitude of the boarding school	double precision
unique_boarding_schools	longitude	Longitude of the boarding school	double precision
unique_boarding_schools	school_type	Type of the boarding school	character varying(20)

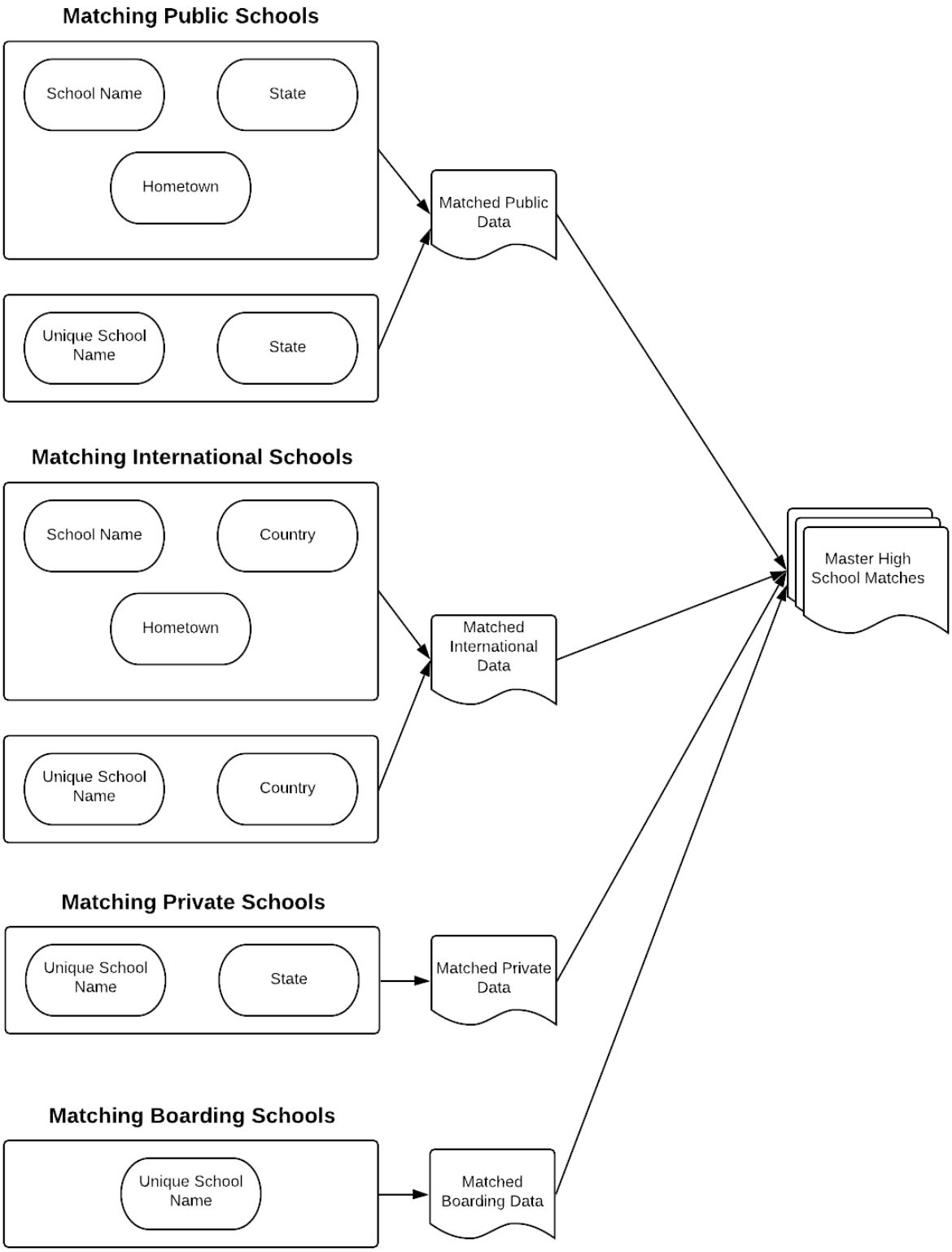
How Matching is Done

Matching must be very carefully done as there is a lot of room for error / false positives. The issue is that the player profiles only provided a hometown and the name of his high school. Since a large number of high schools have the same name, there is no way to guarantee they went to one or the other. We wanted to ensure that we could guarantee 100% matches and not include any data that may be incorrect. So we created strict matching conditions that could guarantee a student went to that school. These conditions involve separating the type of school, for example we know if the player attended a public school, then the high school must be within his/her state boundaries. All matching conditions are displayed in the diagram below. Additionally, we cross referenced unique boarding schools so that we could guarantee matches across state boundaries. Lastly, we looked at the alternative school column and moved the data to the high school column if the high school was blank (null). This was manually done for the original data (roster_data.csv). We encourage looking more into the alternative school column as this can certainly get some more matches as we found out while experimenting with it.

The only way to get more matches with the information provided, is to manually match schools that had slightly different spellings. While future development can explore other matching options, we highly recommend carefully analyzing your techniques to ensure you can guarantee the match. Additionally, adding data in future years will make the software more intelligent, so the best course of action may be to look into scraping more data.

If you wish to see the currently unmatched players to assist with manual uploading you can see a .csv file of them called `unmatched_players.csv` in the `templates` folder in the repository. If you wish to get a new export of the unmatched players use the `unmatched_players` psql script in the `db_info` folder of the repository. You must execute this in parts so follow the comments in the script. Once complete, you can export the results in the roster_master_data table to a .csv file. The code is all in the script as well as a way to count the total number of original players (grouped up). This can be useful when determining how much of the data was captured. Right now this line is commented out. This must be done locally, however, since you cannot copy to a .csv on the server without special permissions. Just ensure you have all the original data in the directory where the script is run and the script will take care of the rest (creating tables, etc.). To run the script in full enter this command:

```
psql < unmatched_players
```


Matching Public Schools
School Name
State
Hometown
Unique School Name
State
Matched Public Data
Matching International Schools
School Name
Country
Hometown
Unique School Name
Country
Matched International Data
Matching Private Schools
Unique School Name
State
Matched Private Data
Matching Boarding Schools
Unique School Name
Matched Boarding Data
Master High School Matches

Dropping and Recreating the Database

Our repository contains a folder called `db_info`. This folder contains scripts to manage the database such as dropping tables, clearing all tables, and recreating the entire database. NOTE: These operations should not be done on the production server. If you wish to go back to an old version use the version control functionality (described in this documentation). This is meant for doing a clean restore of the original database with the original scraped data.

The `drop_all_tables` script deletes and drops all tables, functions, and triggers. This does a full 100% clean and should only be used if you want to re-migrate the database in django or load in a backup.

The `drop_tables_server` script drops all the non-django created tables and clears all the django created tables, so you don't have to re-migrate the database in django. Ideally, this should be used in conjunction with the `entire_job` script.

The `entire_job` script reloads the database doing all the matching, joins, etc. It assumes the django created tables are present so you must ensure the database has been migrated in django. The best way to clear and reload would be running `drop_tables_server` followed by `entire_job`. You must ensure all .csv data found in the `data` folder is present in the same directory you are running the scripts in.

To run any of the scripts login to the server and run the command:

```
psql soccer < [script name]
```

Ex:

```
psql soccer < entire_job
```

Backing Up Database

If you wish to do a general backup of the entire database from the command line (not via the website) run the following command (Note: it can be either a general file or a .txt file):

```
pg_dump soccer > [dump_name]
```

In order to restore the backup, the entire database must be clear, so run the drop_all_tables script to delete all tables, functions, and triggers. Then run the command:

```
psql soccer < [dump_name]
```

Updating Data Conditions

There are two cases on the website where you can update the data.

The first section is 'Upload Scraped Data'. This section is where you would upload newly scraped data. A case of this would be if you scraped new data for 2019 and wanted to upload it. You must upload three files and the column names must match those in the originally scraped .csv file. This is described in the database schema (reproduced again below). Templates for these files are included in the `templates` folder in the repository. Once the data is uploaded the associated tables (map_rosterdata, map_starterdata, map_accoladedata) in the database, we insert into documents, which triggers the following operations: joining new player data with high school, then they are added to the current data and re-grouped up. The `entire_job` script found in the `db_info` folder contains the trigger for an insert on the documents table (and it is commented) if you wish to see this process in more detail. It uses the same joining logic that the original database creation process used.

If you wish to run any test files to test to make sure the 'Upload Scraped Data' is working correctly, there are three test files to upload in the `test_files` folder in the repository. **ONLY UPLOAD SUCH TEST DOCUMENTS IN LOCAL DEVELOPMENT SO BOGUS DATA DOESN'T GO TO PRODUCTION.** To clean the server back to its original state after uploading test documents, run "psql soccer < drop_tables_server" in the ../data directory on 139.147.9.235 then "psql soccer <entire_job"

File Name	Attribute	Description
roster_data.csv	id	Unique id of the player
roster_data.csv	roster_year	Year this player was on the roster
roster_data.csv	player_number	Player's number
roster_data.csv	first_name	Player's first name
roster_data.csv	last_name	Player's last name
roster_data.csv	year	Player's school year
roster_data.csv	position1	Players primary position

roster_data.csv	position2	Player's secondary position
roster_data.csv	position3	Player's tertiary position
roster_data.csv	height	Player's height in inches
roster_data.csv	weight	Player's weight in pounds
roster_data.csv	home_town	Player's home town
roster_data.csv	state_or_country	Player's state (if applicable) or country of residence
roster_data.csv	high_school	Name of the player's high school
roster_data.csv	alternative_school	Name of the player's alternative school
roster_data.csv	college	Name of the player's college
roster_data.csv	college_league	League of the player's college
roster_data.csv	bio_link	Link to the player's college bio
starter_data.csv	id	Unique id of the player
starter_data.csv	roster_year	Player's year on the roster
starter_data.csv	number	Player's number
starter_data.csv	first_name	Player's first name
starter_data.csv	last_name	Player's last name
starter_data.csv	potential_starts	Number of games that the player's team played that season
starter_data.csv	gp	Number of games the player played that season
starter_data.csv	gs	Number of games the player started that season
starter_data.csv	is_starter	Was the player a starter in the given year?

accolades.csv	id	Unique id of the player
accolades.csv	roster_year	Year the player is on the roster
accolades.csv	first_name	First name of the player
accolades.csv	last_name	Last name of the player
accolades.csv	accolade	Accolade the player received that year
accolades.csv	college	College of the player

The second section is ‘Upload Manual Data’. This section is where you would go to upload one matched entry at a time, skipping the joining process. It uploads directly to the map_highschoolmatchmaster table in the database. Then we insert null values to the documents table so we can trigger the grouping to be done. Since it is the same trigger used in the ‘Upload Scraped Data’ section, it will try to rematch, but the scraped data tables (map_rosterdata, map_starterdata, map_accoladedata) are empty since we didn’t insert anything into them (and they are cleaned up after each update) so you won’t get any matches. But this is okay because we bypassed this step since manual matching was done so we shouldn’t be getting any matches. Once this is done the trigger will continue to run as normal and group up the map_highschoomatchmaster table to update map_groupeddata to display the newly updated data.

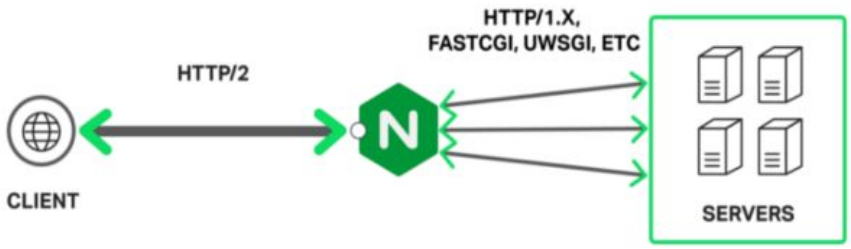
If you wish to see the unmatched players and the list of all high schools (with location info) to assist with manual data upload, you can find them in files called `unmatched_players.csv` in the `templates` directory and `high_school_data.csv` in the `data` directory, respectively.

Running the Server

For the ease of development, we coded locally with our favorite IDEs. When we want to move to the server, we ssh into the server and clone our Git repository there.

In order to have a centralized database during development, we use the database on the server with ssh tunneling to connect to the server from our local projects.

To handle the httpRequest from the user browser, we use a web server software called nginx which serves the static web pages.



To configure Nginx, run the following unix commands.
To install Nginx:

```
sudo apt-get update
sudo apt-get install python3-pip python3-dev libpq-dev postgresql postgresql-contrib nginx
```

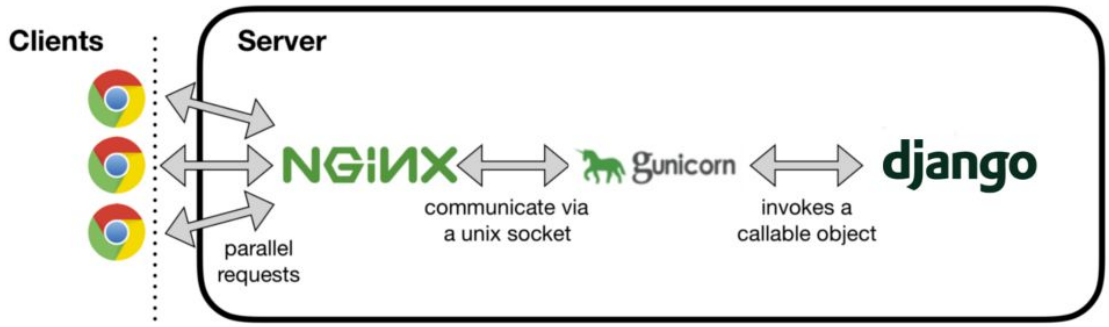
To create and configure Nginx server block:

```
sudo nano /etc/nginx/sites-available/”project name”
server {
    listen 80;
    server_name “server domain”;
    location = /favicon.ico { access_log off; log_not_found off; }
    location /static/ {
        root “project directory”;
    }
    location / {
        include proxy_params;
        proxy_pass http://unix:”Gunicorn sock location”;
    }
}
```

To start Nginx service:

```
sudo ln -s /etc/nginx/sites-available/”project name” /etc/nginx/sites-enabled
sudo systemctl restart nginx
```

To handle the request for dynamic contents, Gunicorn is used to connect the python code through Django wsgi to process the logic, applying filters to the table for example.



To install Gunicorn:

```
pip install django gunicorn psycopg2
```

To create and configure Gunicorn systemd service file:

```
sudo nano /etc/systemd/system/gunicorn.service
[Unit]
Description=gunicorn daemon
After=network.target
[Service]
User=”user”
Group=www-data
WorkingDirectory=”working directory”
ExecStart=”wherever gunicorn is in your system”/bin/gunicorn --access-logfile - --workers 3
--bind unix:”your project location/project”.sock myproject.wsgi:application
[Install]
WantedBy=multi-user.target
```

To start the Gunicorn service:

```
sudo systemctl start gunicorn
sudo systemctl enable gunicorn
```

To secure our sensitive information from data transmission, we used Certbot to get our site on https.

To configure Certbot on the server:

```
sudo apt-get update
sudo apt-get install software-properties-common
sudo add-apt-repository universe
sudo add-apt-repository ppa:certbot/certbot
sudo apt-get update
sudo apt-get install certbot python3-certbot-nginx
sudo certbot --nginx
```

Extending the Application

This section details suggestions in how we believe the application can be extended and improved upon in the years to come.

Scraping New Player Profiles

As we detailed in our Database section, we have a very strict matching policy with Players and High Schools to ensure the data can be 100% confirmed to be correct. So since matching really can't provide the application with any more data, the best way to increase the intelligence of the application would be to upload new scraped data (we stopped at 2018). The `scripts` folder of our repository contains the code for how Cameron Zurmuhl originally did the scraping from the college sites. This code would have to be modified to account for the changed sites and output into three files: roster_data.csv, starter_data.csv, and accolades.csv. These would be uploaded to the 'Upload Scraped Data' page of our site. The column names in these files must be an **exact match** to the originally scraped files (and the associated tables in the database) as described in the 'Updating Data Conditions' section of this document (explained in greater detail and displays the schema).

Another nuance to consider is integrating alternative schools into the matching process. A possible solution is if the high school is null, add in the alternative school as the high school.

Refining Version Control

Another way this application can be extended would be to refine when you can save versions of the data on the website. Right now a backup (dump file) is created in the map_backup table of the database every time new data is uploaded via 'Upload Scraped Data'. However, there is no such backup functionality for the 'Upload Manual Data' page. Since it would be inefficient to create a backup each time you added one entry here, we would have liked to add a save feature that lets you create a backup after you are done manually uploading a few entries (or whenever you are content to do so). Implementation would be similar to how we do it after scraped data is uploaded, except you would need some sort of a save button that triggers this action on the 'Upload Manual Data' page.