Technical Report

Tyler Pak, Tyler Black, Jack Burrus, Lucy Zheng

Team E8

Men's College Basketball Internet Database

Contact List:

Tyler Pak (led phase 1): paktyler@utexas.edu, Github username: tylerpak

Tyler Black (led phase 2): tyler.black@utexas.edu, Github username: guest1GRjBAYo

Lucy Zheng (led phase 3): lucy.zheng@utexas.edu, Github username: lzheng777

Jack Burrus (led phase 4): jackburrus@utexas.edu, Github username: jburrus1

Github repo: https://github.com/UT-SWLab/TeamE8

Deployed app: https://college-basketball-infosite.uc.r.appspot.com/

MOTIVATION AND USERS

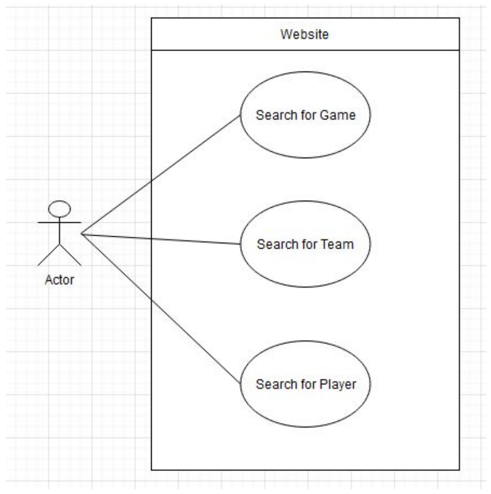
We picked college basketball as our topic because we saw great potential for an application that can help users follow the sport, make predictions, or build fantasy teams. We want users to be able to find all the information that they would need in order to follow the sport or participate in fantasy college basketball. We expect our users to be casual basketball fans who enjoy watching the sport, and now want to learn more about it. Because of this, we want our website to be easy to navigate and interpret the information for a new or casual basketball fan.

USER STORIES

PHASE 1

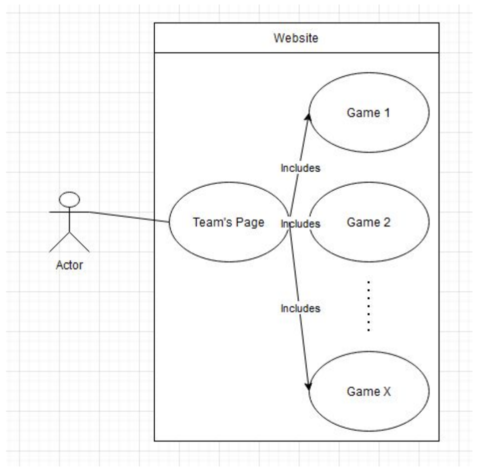
1. As a user, I can search for teams, players, or games by name or date the game was played. Users on our site will be able to search for instances of our models and select specific instances with ease.

Completion Times: Estimated 3hrs, Actual 4hrs



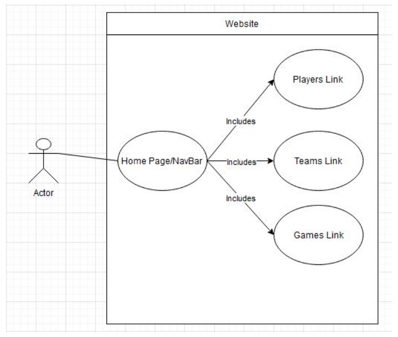
2. As a user, I can access the games a certain team participated in from their instance page. Users on our site can currently see “featured games” on each team page and select one to take them to the corresponding game page.

Completion Times: Estimated 1hr, Actual 2hrs



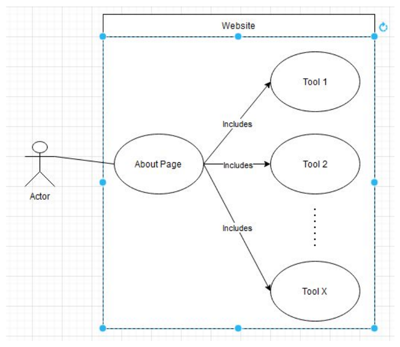
3. As a user, I can access a home page that contains navigational links to other pages. Users will initially arrive at a home page that gives them access to every other page on the site via links and navbar.

Completion Times: Estimated 2hrs, Actual 2hrs 30min



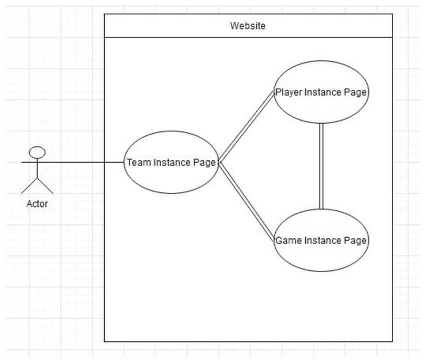
4. As a user, I can view the tools and APIs used on the site. Users can access and "About" page that includes all the tools we used, and links to all our APIs.

Completion Times: Estimated 30min, Actual 30min



5. As a user, I can access other instances from an instance page. Users can visit instances from other models from an instance page. For example, a user that is on a player instance page, can access that player's team page, or a game page that the player participated in.

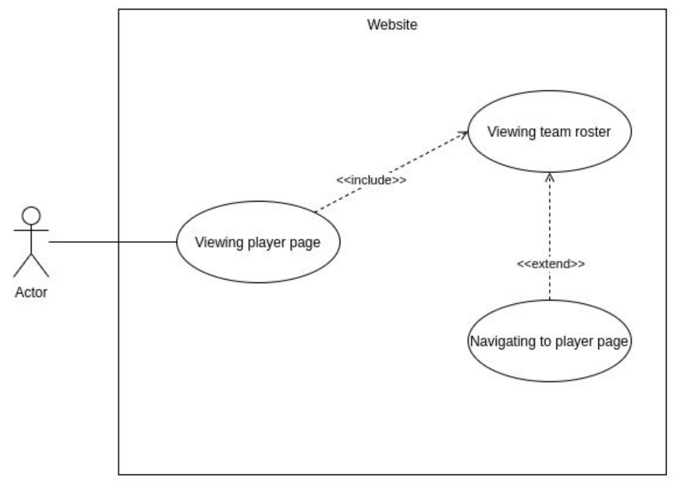
Completion Times: Estimated 1hr, Actual 1hr



PHASE 2

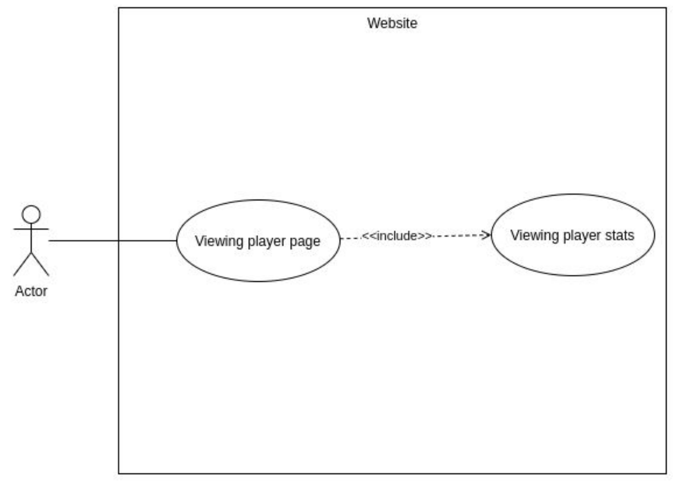
1. As a user, I want to be able to see the entire team's roster when on a team's page. These players should appear as names and I should be able to access that player's page by clicking on their name.

Completion Times: Estimated 1hr, Actual 1hr



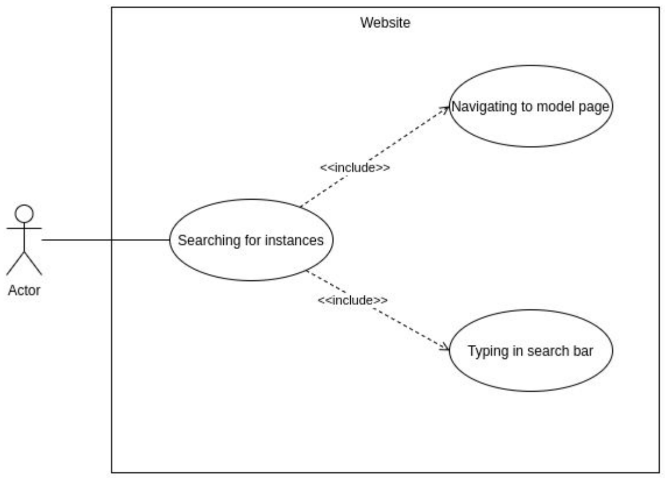
2. As a user, I want to see individual player stats on each game page. These stats should appear in a readable manner and be relevant to the subject

Completion Times: Estimated 30min, Actual 1hr



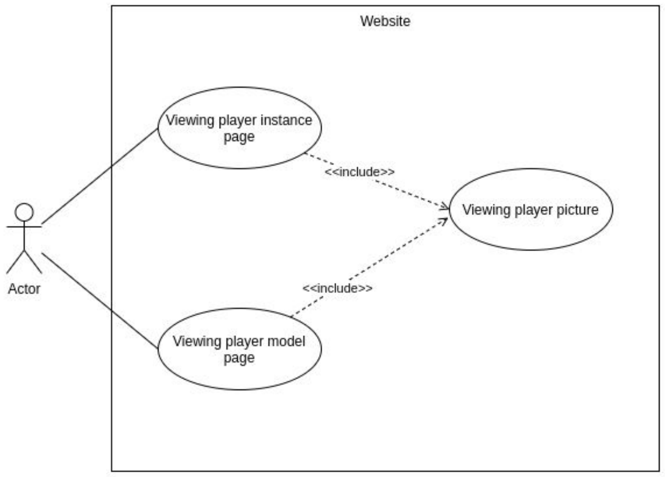
3. As a user, I want to search instances by name. There should be a search bar where I can search for instances easily and get results in a usable way.

Completion Times: Estimated 2hrs, Actual 2hrs 30min



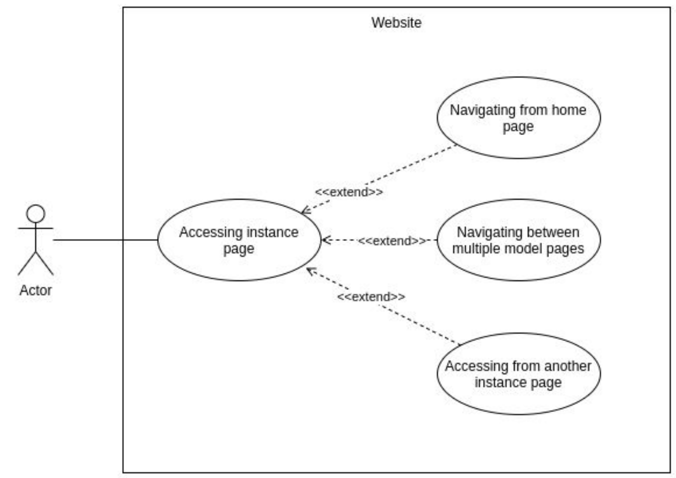
4. As a user, I want to be able to see players' profile pictures. These pictures should be visible and displayed in an appealing manner.

Completion Times: Estimated 30min, Actual 30min



5. As a user, I want to be able to navigate the site intuitively. This means I should be able to access all model pages easily and move through results using pagination.

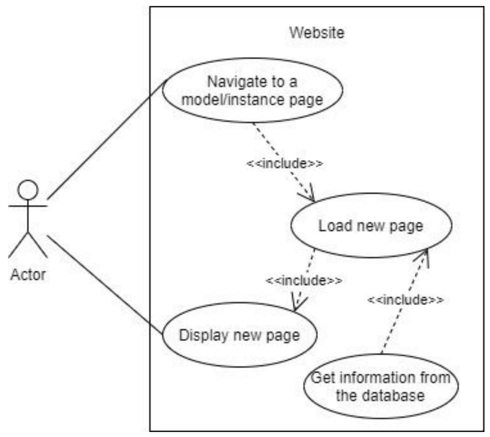
Completion Times: Estimated 1hr, Actual 2hrs


Website
Navigating from home page
<<extend>>
Accessing instance page
<<extend>>
Navigating between multiple model pages
Actor
<<extend>>
Accessing from another instance page

PHASE 3

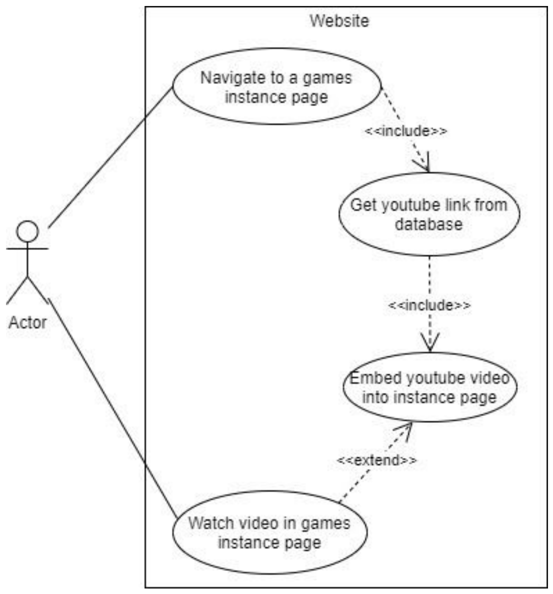
1. As a user, I want the load times for each page to be faster. This means that navigating to a different page has a small number of database accesses.

Completion Times: Estimated 2hrs, Actual 1hr 30min



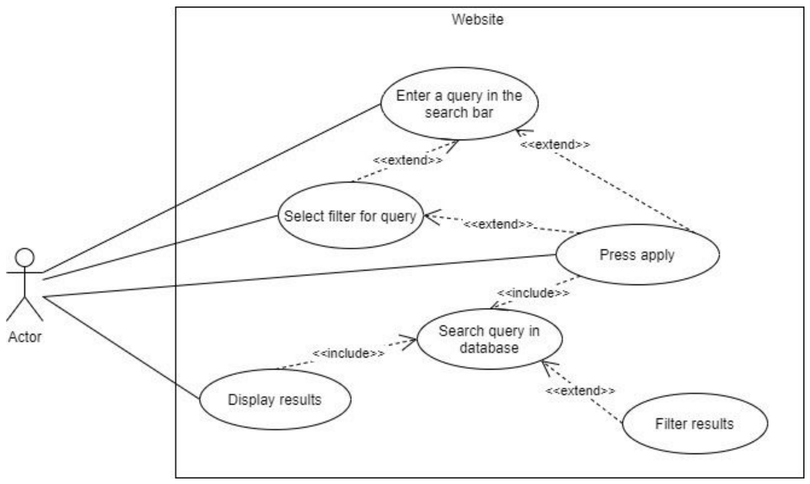
2. As a user, I want to see the highlights for each game in the database. This means that there should be a video embedded into each of the game instance page.

Completion Times: Estimated 2hrs, Actual 3hrs 30min


Website
Navigate to a games instance page
<<include>>
Get youtube link from database
<<include>>
Embed youtube video into instance page
<<extend>>
Watch video in games instance page
Actor

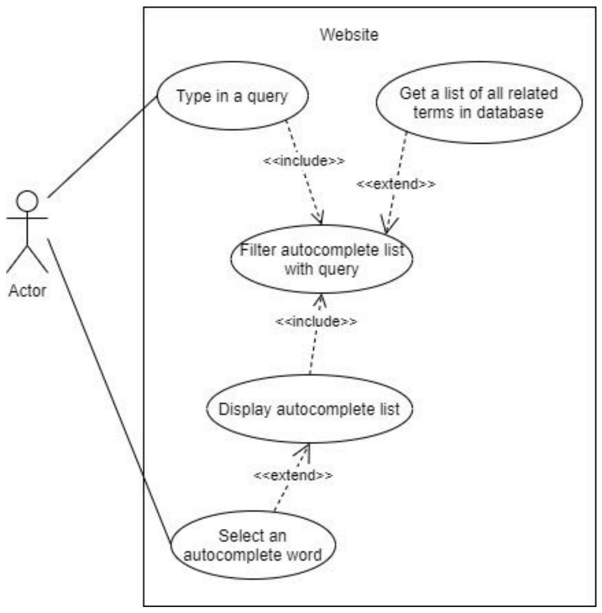
3. As a user, I want to be able to filter my search results. If I enter a keyword into a search bar, I should be able to narrow down the results in order to find what I am looking for much easier.

Completion Times: Estimated 3hrs, Actual 4hrs



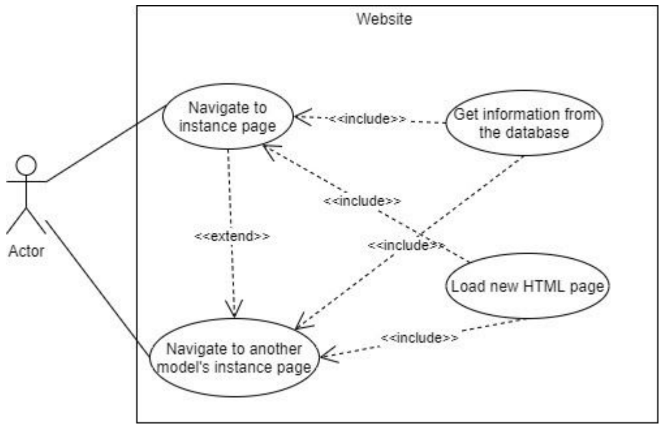
4. As a user, I want my query to be autocomplete whenever I type something into the search bar. This means that if the search bar is not empty, it gives me a list of suggested terms that match what I have typed into the search bar.

Completion Times: Estimated 2hrs, Actual 2hrs 30min


Website
Type in a query
Get a list of all related terms in database
<<include>>
<<extend>>
Actor
Filter autocomplete list with query
<<include>>
Display autocomplete list
<<extend>>
Select an autocomplete word

5. As a user, I want the instance pages to be more uniform with the rest of the website pages. This means that each page looks similar to other instance pages and the website looks uniform.

Completion Times: Estimated 3hrs, Actual 3hrs


Website
Navigate to instance page
<<include>>
Get information from the database
<<include>>
Actor
<<extend>>
<<include>>
Load new HTML page
<<include>>
Navigate to another model's instance page

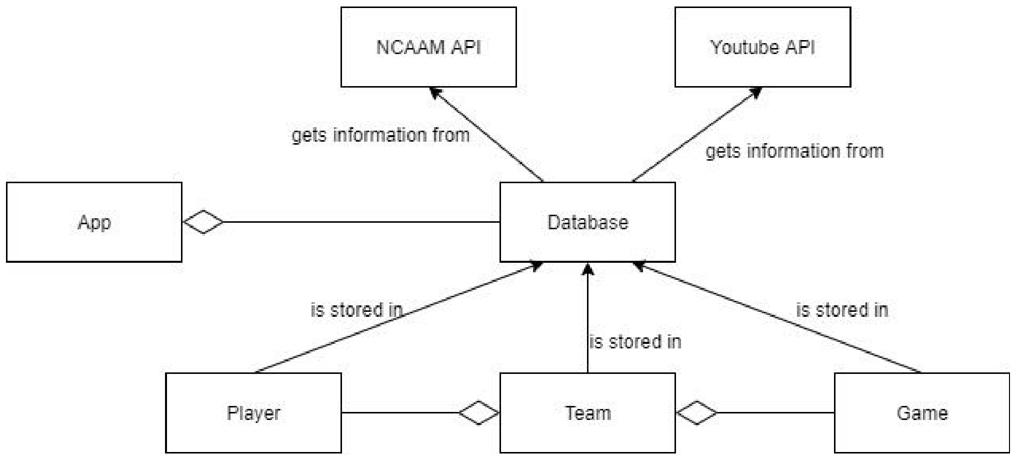
DESIGN

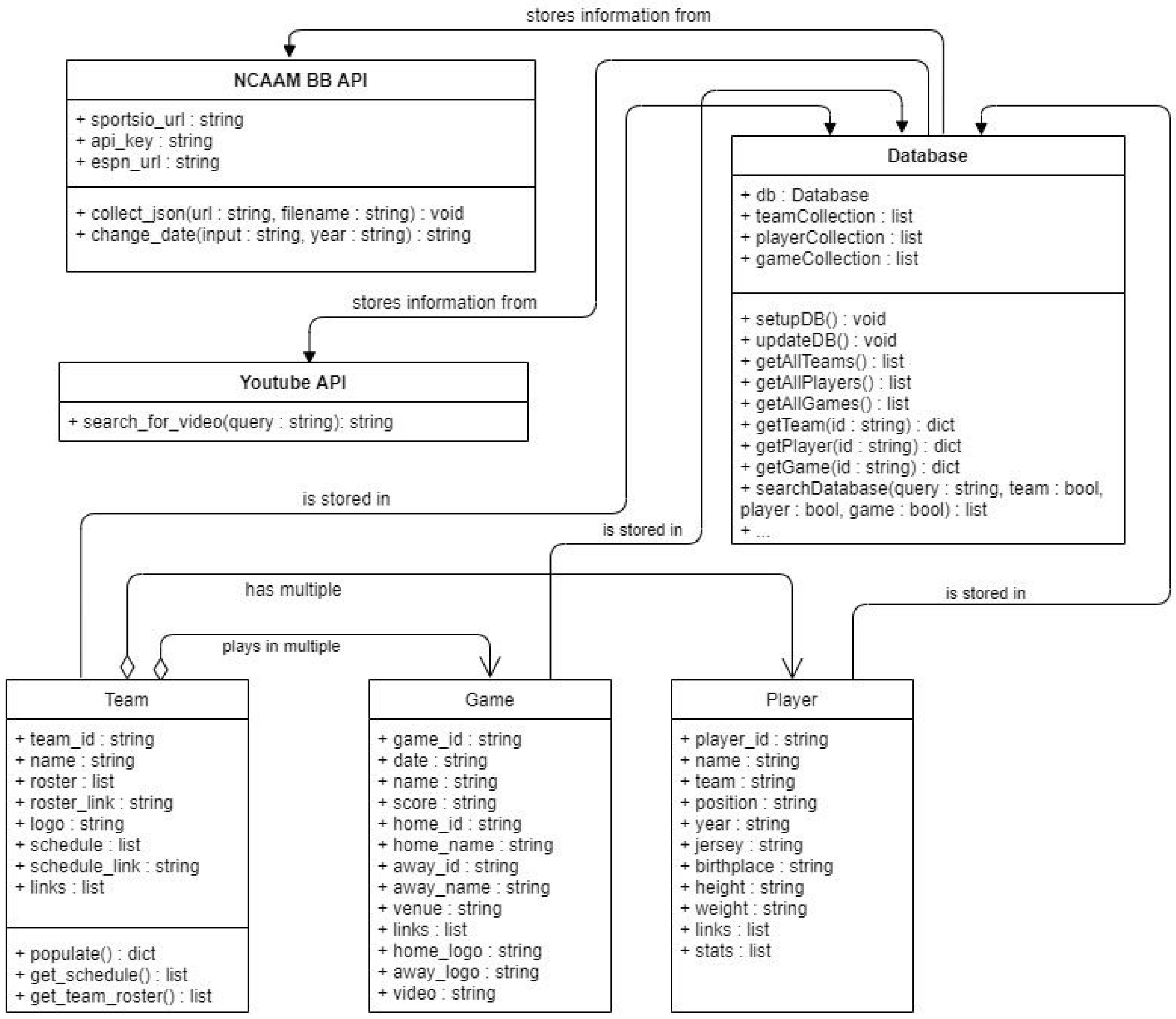
As shown in the UML class diagram below, we created a total of 6 classes to get and store data from the APIs we used. The NCAAM BB API gets all the related information from the API and organizes the data into Team, Player, and Game objects that the database uses. The Youtube API searches Youtube with the name of the game and returns the first result of the query. The database class uses both API classes to populate and store all Team, Player, and Game object information.

The database class has a total of 5 collections to organize the stored information: Teams, Players, Games, News, and Autocomplete. All Team, Player, and Game information is stored in the database and can be returned as a result of a query. The database implements the singleton design pattern so that there is only one Database object to prevent concurrent database access and modification. As a result, all database related functions are in the Database class. The collections also belong to the Database class to prevent any other module from accessing any database resources without using the single Database instance.

The Team, Player, and Game objects contain information from the API and multimedia links. Each Team has a list of Games and distinct players. Each game is linked to 2 teams, the home team and the away team, and indirectly to the respective teams' rosters. Each player is linked to a single team and a list of games that their team was scheduled to play in, whether they were actively in the game or not. The Game and Player class do not have a direct field related to each other, but are connected via their teams, with the team rosters being listed on game pages and team schedules being listed on player pages.

The App handles all redirects and requests from the HTML files. The App has a Database object to retrieve information from the database and pass it to the HTML files. This design allows the Database object in the App to handle all database requests and makes the database in charge of organizing it appropriately so the App just needs to pass the data on to the page.


NCAAM API
Youtube API
gets information from
gets information from
App
Database
is stored in
is stored in
is stored in
Player
Team
Game


stores information from
NCAAM BB API
+ sportsio_url : string
+ api_key : string
+ espn_url : string
+ collect_json(url : string, filename : string) : void
+ change_date(input : string, year : string) : string
stores information from
Youtube API
+ search_for_video(query : string): string
Database
+ db : Database
+ teamCollection : list
+ playerCollection : list
+ gameCollection : list
+ setupDB() : void
+ updateDB() : void
+ getAllTeams() : list
+ getAllPlayers() : list
+ getAllGames() : list
+ getTeam(id : string) : dict
+ getPlayer(id : string) : dict
+ getGame(id : string) : dict
+ searchDatabase(query : string, team : bool, player : bool, game : bool) : list
+ ...
is stored in
is stored in
is stored in
has multiple
plays in multiple
Team
+ team_id : string
+ name : string
+ roster : list
+ roster_link : string
+ logo : string
+ schedule : list
+ schedule_link : string
+ links : list
+ populate() : dict
+ get_schedule() : list
+ get_team_roster() : list
Game
+ game_id : string
+ date : string
+ name : string
+ score : string
+ home_id : string
+ home_name : string
+ away_id : string
+ away_name : string
+ venue : string
+ links : list
+ home_logo : string
+ away_logo : string
+ video : string
Player
+ player_id : string
+ name : string
+ team : string
+ position : string
+ year : string
+ jersey : string
+ birthplace : string
+ height : string
+ weight : string
+ links : list
+ stats : list

TESTING

SELENIUM TESTS

There are four basic tests for website navigation as well as three more complex tests. In addition, there are a total of four tests for search, filter, and sort functionality.

The basic tests, test_about(), test_playerModel(), test_teamModel(), and test_gameModel(), test navigation using the navbar to the about page, 1st player page, 1st team page, and 1st game page respectively. Since the navbar is inherited by every page, the tests all run from the home page, but should work regardless of the page they start on.

The complex tests, test_playerInstances(), test_teamInstances(), and test_gameInstances(), test every link on each player, team, and game instances on the first page of each model to make sure there are no broken links. Ideally, these would check every instance page, but currently the tests take a very long time to run, so we limited them to the first pages of each model.

There is one test for searching, test_mainSearch(), which searches for "a" on the main page because it is guaranteed that instances of all types exist containing the letter a. After that, it searches for a string of nonsense characters to test for a scenario in which no results are given. It repeats this process for each of the 3 models. The three filter tests each select each filter and sort option before applying it, testing that each one results in a valid page.

UNITTEST

The API was tested using Unittest in python. The API sends HTTP requests to the ESPN database and receives back JSON data. The API has some built in error handling so failures are less likely to occur. The API uses try/except blocks to catch potential faulty requests or faulty responses. The testing script further reinforces this by asserting the type that each value returns. The API has four classes: Players, Games, Teams, and News. These classes are primarily a collection of attributes. These attributes are defined inside of the try/except blocks, so they get properly defined even if they don't receive the proper information. The testing script uses a sample set of each class to test the validity of each class's attributes. This allows us to have confidence that the API returns valid data, even if it is not accurate. For example, the Team class has a record attribute that has been returned empty because the 2020 season was canceled. We expect that attribute to start being filled here in a couple weeks as the 2021 season gets underway and we also expect the API to be able to handle this change properly due to the testing that we've put it through.

MODELS

Our models are players, teams, and games. The Team model represents NCAAM basketball teams. The model contains team rosters, schedules, name and logo as its attributes. Teams link to collections of players, through the roster, and to collections of games, through the schedule. A team can be searched by name or players through the search bar. Teams can be sorted alphabetically.

The game model represents a contest between two NCAAM teams.The game model contains a title, score, venue, relevant media, and the two opposing teams and their players. The game models link to the instance pages of the players and teams who participated. Games can be searched on the search bar by title, teams or players participating, venue or date. Games can be sorted by home or away team alphabetically, or by venue alphabetically.

The player model represents players that play on teams in the NCAAM league. The player model contains player name, position, hometown, jersey, statistics, height and weight, team, profile picture, and games participated in. The player model links to games through their team's schedule and is linked to the team that the player plays for. Players can be searched by name, team, position, hometown, or height and weight through the search bar. Players can be filtered by position and sorted alphabetically by name or team, or by increasing or decreasing weights.

TOOLS

MongoDB- Database used to store all information collected from the APIs we used. There is 1 database with 5 collections to store the collected information from the ESPN API and the YoutubeSearch API. The database organizes the information into 1 of 4 collections: Teams, Players, Games, or News. The 6th collection stores information for autocompletion and related terms.

Bootstrap4- Bootstrap was used for designing the website and making it responsive. We made extensive use of bootstrap div classes, especially containers, rows, and columns. Many of the attributes we wanted to display on instance pages lent themselves well to a grid or list display, so bootstrap was used to make sure that things were displayed properly. We also used bootstrap for the creation of the navbar, which is displayed on every page and changes functionality based on screen size, taking advantage of bootstrap's responsive classes.

YoutubeSearch API- The youtube API was used to supplement our primary API with more multimedia. The ESPN API gives us links to other websites for multimedia at best. We used the youtube API to search for game highlights for the Game() class. The API searches youtube with a query containing both team names and the date of the game and returns the top result. For Power 5 conference teams, this worked very well; however, smaller, non-Power 5 schools did not always return the correct game highlights.

ESPN API- The ESPN API, though not officially supported, served as our primary data source. They have tons of data on teams, games, players, and news. The API excelled in giving us data on our classes, but was lacking in multimedia. Since the API is not officially supported, there is no current documentation for it; therefore, working with it was an exercise of trial and error in terms of finding endpoints. Over the course of the semester, however, the API has proven robust and reliable.

SportsDataIO API- An API used for collecting more data on players specifically. We wanted to supplement our primary database with more data, especially player background and statistical data. Sports Data gave us player background information such as hometown, birthplace, etc. It also gave us better statistical information for a given player, in terms of game, season, and career.

Flask- Used for site framework and adding some functionality to the website. We used Flask to also handle interaction between our website and users through the “form” class and methods GET and POST, mainly utilized in the implementation of our search bar. Some functionality was also added to pages using Flask, including the ability to check if players and games exist in the database, which is useful for making sure that no broken links are left on pages. Flask also comes with Jinja functionality, which was used to dynamically add to pages. Using Jinja, we could display lists of information from the database, show grids of instances on the model pages, and store variables for use later, such as the page name. Jinja also allowed for the use of templates, such as our base template that was extended by every page, giving each page a navbar as well as colored sides for styling.

CSS- Used for styling the website to make it look nice and to have a consistent look across the site. We used a base CSS file to handle overall styling, and each html page had some CSS of their own for tailored styling to the page. We made use of class selectors, especially in conjunction with Bootstrap, to specify the styling for certain parts of pages.

Google Cloud- Google Cloud is the platform chosen to host the website. The platform is also used to check the status of the current deployment and if there are any issues, such as a bad gateway or long latency times.

Selenium- Selenium was used for front-end testing. Most of the tests involved site navigation, and later tests expanded to the testing of searching and filtering on model pages. More information on selenium can be found in the testing section of the report.

Git/Github- Used for version control and sharing code. Github was used to handle most of our project management. We made project boards to handle task tracking for phases 1-3 and organizing our user stories. All issues were opened and handled on github.

Slack- Slack was used as our main method of communication throughout the project. We used three channels to organize communication: #general, #important_links and #updates. The #general channel was used to handle all discussion about the project. This was a place where members could discuss ideas and ask questions. We used the #important_links channel to carry and organize all shared documents and links so any member could quickly access necessary resources. Lastly, the #updates channel is where each member posted their updates. These updates would be posted about three times a week by each member, and would include status on current and upcoming tasks, and any issues or complications in work.

Sources used for references:

https://www.w3schools.com/bootstrap4/

Bootstrap tutorial used for frontend design.

https://www.tutorialspoint.com/mongodb/index.htm

Mongodb tutorial to help with database development.

APIs:

https://github.com/md130330/Ncaabballstats

NCAA BBall Stats provides extra player statistics, per game.

https://sportsdata.io/

Sports Data IO gives fantasy data for players and teams.

http://www.espn.com/apis/devcenter/docs/

ESPN provides most of our data for Teams, Players, and Games.

REFLECTIONS

PHASE 1

Our team learned a lot in phase 1. Tyler Black learned about HTTP requests and json formatting and accessing. He learned that HTTP requests take much longer than any other part of the API so he realized that optimizing the speed of the API was something he would need to focus on. Lucy learned how to write functions that return data from the API. Jack and Tyler Pak learned how to use bootstrap to create responsive websites that are easy to navigate. We also learned a lot about each of our individual strengths and weaknesses as programmers, and how to work with each other to create a product.

Five things that we struggled with and need to improve:

1. API speed optimization when dealing with HTTP requests
2. Database organization with MongoDB
3. Presentation of information in a visually appealing manner on the website
4. Supplementary data and information for webpages
5. Reviewing each other's work

Five things that worked well:

1. Communication was great between all team members; we all knew what each of us were responsible for
2. Creating the basic website components
3. Deploying the website
4. Team meetings were very efficient and deliberate
5. Individual initiative; all team members constantly looked for ways to improve the product throughout development

PHASE 2

Our team refined knowledge in phase 2. Tyler Pak used jinja2 to display data from the database to the webpage. Jack learned how to test website navigation with selenium and realized that selenium is intuitive once he understood it. Lucy struggled with writing the database at first, but learned MongoDB through various online resources. Tyler Black learned a lot about regex in this phase and used it to get data that not was not officially offered by the API.

Five things that we struggled with:

1. Initial understanding of the tools we needed to use
2. Initial time management. A lot of us were busy early on with other classes
3. Clear understanding of specific requirements of product
4. Page loading times
5. Efficiency of code in certain methods

Five things that went well:

1. Communication between frontend and backend. All data that we needed was easily transferred from API to database to webpage
2. Communication between team members was great
3. Work ethics. All team members put in a good deal of work and carried their weight
4. All tests provided good results in terms of site functionality and navigation
5. Team meetings were efficient and purposeful

PHASE 3

This section we all continued to learn more about our respective responsibilities. Jack learned a lot more about bootstrap when working on the styling of instance pages, especially bootstrap grids. Updating selenium tests went very well with Jack because of how intuitive it is to test with. Tyler Pak struggled at first to implement filtering, sorting and searching features to the model pages, but learned that you can pass multiple variables through a url which came in handy when trying to keep track of a query, filter and sort value. Lucy learned a lot about handling HTML and Javascript when implementing the autocomplete for the search function. This was sort of new since Lucy has been working pretty exclusively on backend for the most part. Tyler Black learned a lot about exception handling when dealing with invalid data coming from our APIs.

One thing that we struggled with is that our youtube search requests returns a video that we hope is related to a specific game based on the keywords we used in our search request, but there is no way to be sure. That means that some videos on our game instance pages are not related to the game instance. This is something we will have to focus on and correct in the next phase.

Five things we struggled with:

1. Adding the correct youtube videos to game instance pages
2. Issues with database for unknown reasons
3. Finding filterable attributes
4. Loading times
5. Getting enough news articles for team instances

Five things we did well with

1. Search and autocomplete functionality
2. Improved styling
3. Tests provided good results
4. Communication
5. Sorting results

PHASE 4

In our final phase of this project, our team practiced refactoring and design pattern implementation. Jack focused on describing the modularization of our project which required him to review information hiding concepts. Lucy put her efforts in implementing the singleton design pattern in our project. Tyler Black focused on refactoring his API methods to improve code styling and form. Tyler Pak worked on improving the technical report from our previous submission.

Five things we struggled with:

1. Understanding phase requirements

2. Time management with Thanksgiving break
3. Slow communication at the beginning of the phase
4. Focusing more on code styling and modularization earlier in the project would have made phase 4 easier to implement
5. Expanding technical report based on homework 5 feedback

Five things we did well with

1. Task organization
2. Implementing design pattern to project
3. Communication later in the phase
4. Efficient work in all areas
5. Structuring of design report