DATE:	Nov 30, 2020
TO:	Dr. Eberlein, EE 461L TAs
FROM:	Kevin Medina, kvnmedina59@gmail.com, Kvnmedina
	Samuel Yeboah, syeboahjr@utexas.edu, Samuel-Akwesi-Yeboah
	Alan Zhang, alan.zhang.2000@utexas.edu, adz00
	Edie Zhou, edie198@gmail.com, edie-zhou
SUBJECT:	Phase 4 Technical Report
PROJECT:	Formula 1 Internet Database
GITHUB:	https://github.com/UT-SWLab/TeamE2
CANVAS TEAM:	Team E2
WEBSITE LINK:	https://f1stat-292509.uc.r.appspot.com/

INTRODUCTION

This document serves as documentation for the development process of Team E2's EE 461L project, a Formula 1 Internet Database. The report includes information on the intended users, features, design, testing methods, data models, and tools used.

MOTIVATION

The purpose of the site is to display Formula 1 statistics, aggregated from multiple public APIs. We knew how dominant the industry leaders are in the Formula One space and we wanted to share this with others. While we knew Lewis Hamilton, and Mercedes are by far the best driver and constructor respectively, many other people have no idea of the utter dominance that is currently on display by this driver and constructor. The site features historical and current data for models ranging from drivers to constructors to circuits. The goal of the website is to combine all these components into one easy to access site. Users that would benefit from this site include Formula 1 team members, Formula 1 fans, sports analysts, or anyone else looking to improve their understanding of Formula 1. Additionally, any other developers can utilize our collected data sources should they want to create their own database.

REQUIREMENTS

The team utilized various project planning methods to ensure an organized development. By planning out the project in a careful manner, the team was able to ensure all required features were included on the website.

Phase 1

Phase 1 involved organizing the team compact, discussing possible ideas for the site, and laying out the groundwork for the project development. It included the initial, static creation of the website, with some preliminary data collected in order to generate page examples for the various data models.

User Stories

“As a user, I want to view a racer’s win history, team, age, and all personal information.”
This story was completed by Edie, with an estimated time of 3 hours and an actual completion time of 2 hours. We want to display all driver data on a singular page so that users have a singular location to view statistics about drivers. All of the driver information is stored in the drivers collection in the database.

“As a user, I want a home page so that I can navigate between all of the various pages that are available.”
This story was completed by Sam, with an estimated time of 3 hours and an actual completion time of 2 hours. The homepage is the landing screen that connects all of the individual model pages and instance pages together. The home page also shows some relevant statistics like drivers born today or upcoming races.

“As someone with poor eyesight, I want the user interface to be simple and easy to navigate.”
This story was completed by Sam, with an estimated and actual completion time of 2 hours. We designed the UI to be simple and clear so that users would not have any trouble navigating through the website. We also chose colors that were easy on the eyes and a high contrast so that there would be a high readability level on the pages.

“As a user, I want to view a constructor’s current and former team members, race statistics, and nationality.”
This story was completed by Kevin with an estimated time of 4 hours and an actual completion time of 5 hours. The constructor instance pages show all relevant data regarding the constructor’s or race teams. We assumed that the constructor data would be easy to collect. However, it was difficult to match the drivers to their teams. Also, the driver data is contained in the drivers collection which increases the time it takes to load the constructor pages.

“As a user, I want to see some general information on each driver before clicking on their card so that I don’t need to go all the way to their instance page.”
This story was completed by Alan, with an estimated time of 2 hours and an actual completion time of 3 hours. We want to display some relevant information about each driver on the model page. Users can view important information about the driver before clicking on the specific instance.

Phase 2
Phase 2 saw an increased number of model instances, the use of pagination on the model pages, the collection and calculation of additional data for each instance page, and the optimization of the database structure and the accompanying queries. In this phase, the team eliminated all instances of hardcoded data and saw the creation of the first frontend/backend tests.

User Stories

“As a user, I want to see the past teams that a driver has driven on so that I can see their career progression.”
This story was completed by Alan, with an estimated time of 2 hours and an actual completion time of 4 hours. We wanted to show a driver’s career history on their specific instance page. We assumed we could query the results collection to view the constructors however we were forced to add another attribute to the driver model. By adding an attribute, we were able to decrease the loading time of the driver instance page.

“As a user, I want to see the number of wins for a constructor or a driver so that I can determine who is a better racer and what teams are the best.”

This story was completed by Kevin, with an estimated time of 4 hours and an actual completion time of five 5 hours. We calculated the wins for drivers and constructors by querying the results table and counting the number of 1st place finishes each team had.

“As a developer, I want pictures that accurately represent each instance.”
This story was completed by Edie, with an estimated completion time of 2 hours and an actual completion time of 8 hours. We harvested the pictures from Wikipedia however the picture needed to be cleaned and scrapped manually which took a large amount of time.

“As a developer, I want the ability to test my code so I know the software is free from bugs before I push to production.”
This story was completed by Sam, with an estimated time of 4 hours and an actual completion time of 4 hours. The repository was tested with unittest for the flask functions, and selenium for the frontend. On the flask server, we tested the CRUD operations for each collection that data was pulled from.

“As a developer, I want to be able to test my UI so that I know it is bug-free before I push to production.”
This story was completed by Alan, with an estimated and actual completion time of 2 hours. Selenium was used to test the UI. The test consisted of traveling through multiple pages of the application to make sure the pages rendered correctly.

“As a user, I want to view the web application's data sources and method of retrieval. This is for my own verification and assurance.”
This story was completed by Sam, with an estimated and actual completion time of 3 hours. We wanted to verify our data sources so that we had no bugs in our software.

“As a user, I want to see a short bio/summary of each instance to get some general knowledge on them.”
This story was completed by Kevin, with an estimated time of 1 hour and an actual completion time of 3 hours. Because our API’s did not provide us with a bio, we scraped the biological information about each model instance from the associated Wikipedia page that was linked to in the database. We stored the biological information in an attribute field called bio in each model.

“As a user, I want to navigate through the model instances using an intuitive feature.”
This story was completed by Alan, with an estimated time of 3 hours and an actual completion time of 4 hours. At first, the model pages simply displayed every single model instance on a singular page. However, this caused the page to load extremely slowly. So, we added pagination which limits the number of visible instances to a specified number per page.

“As a user, I want to see the fastest laps of all time on a circuit to see what kind of speed the drivers are able to accomplish.”
This story was completed by Kevin, with an estimated and actual completion time of 2 hours. We calculated the fastest lap time by querying the results database with the circuit ID then sorting the results based on the lap time.

Phase 3

Phase 3 was largely concerned with implementing search, sort, and filter functionalities to the model pages, while still preserving pagination. This phase also saw the refinement of tests, an update to make the homepage cleaner and more interactive, and an update to the instance and homepage CSS to make them mobile responsive.

User Stories

“As a user, I want an interactive homepage that gives me some relevant information so that I can browse the site more easily.”
This story was completed by Sam and Kevin, with an estimated time of 2 hours and an actual completion time of 4 hours. Up until now, the homepage had been left in a state of “work in progress”, but now it features a cleaner design and interesting data such as popular drivers, popular circuits, and drivers born in the current month. It also features upcoming races, driver season standings, and constructor season standings.

“As a user, I want to be able to sort model instances so that I can find the information that I want more easily.”
This story was completed by Alan and Edie, with an estimated time of 3 hours and an actual completion time of 3 hours. The user was given the choice of using three different sort techniques: relevance, alphabetical A-Z , and reverse alphabetical Z-A. This allowed the user to

change the ordering of the instances on the model page based on whatever would be most convenient for them.

“As a user, I want to be able to search model instances so that I can find the information that I want more easily.”
This story was completed by Edie and Alan, with an estimated time of 6 hours and an actual completion time of 4 hours. This gave the user the ability to search our vast collection of drivers, constructors, and circuits for a specific model instance. The search function did its best to return a best match, based on using a “contains” method of searching for whatever terms were entered in the search bar.

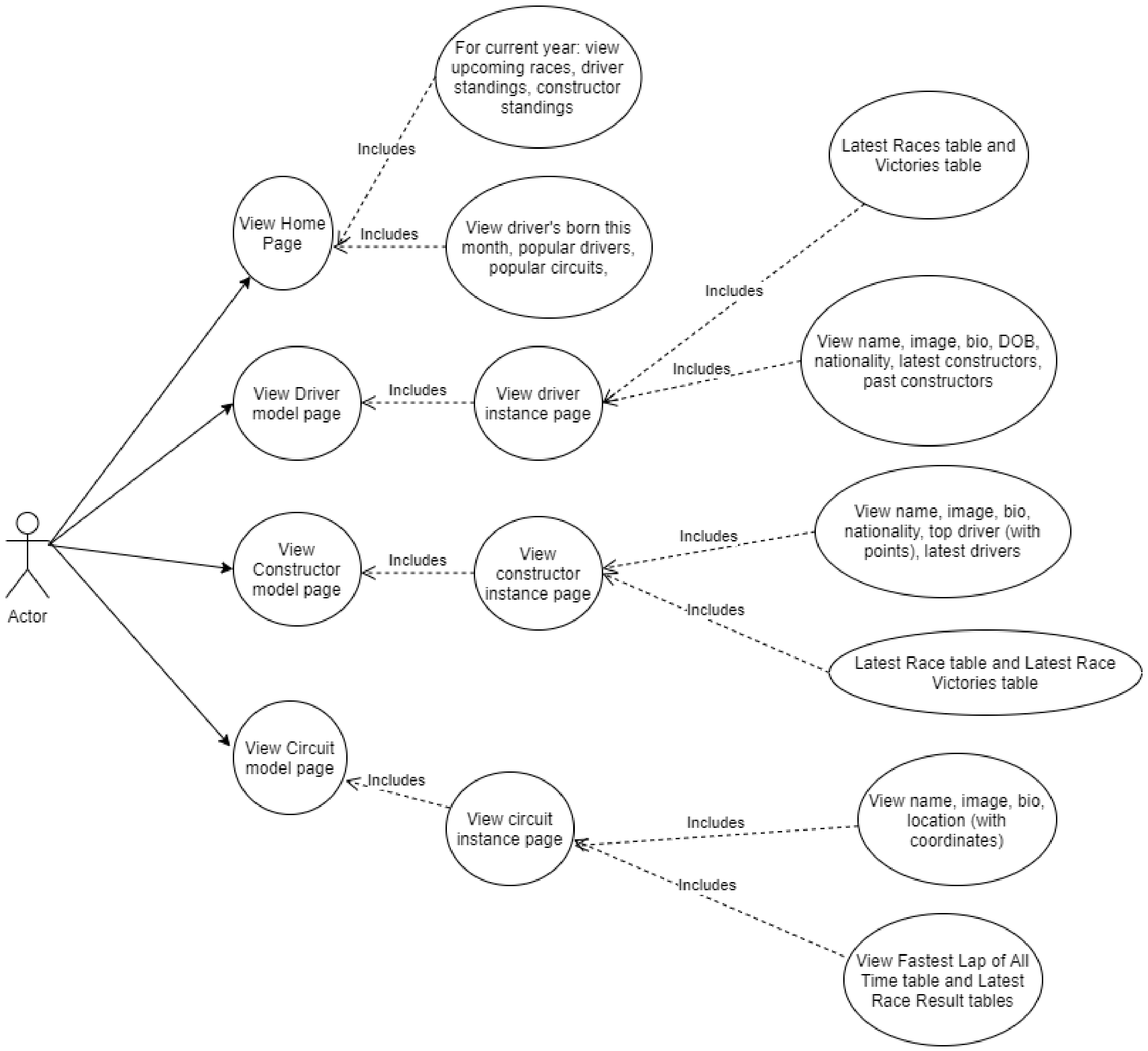
“As a user, I want to be able to filter model instances so that I can find the information that I want more easily.”
This story was completed by Edie and Alan, with an estimated time of 3 hours and an actual completion time of 3 hours. Each model page featured three filter choices, with the first being “name”, and the next two being model specific. This allowed users to focus their search and sort on a specific criterion of the model.

“As a user, I want the website to be responsive to my screen size, even if it is very small.”
This user story was completed by Kevin, with an estimated time of 3 hours and an actual completion time of 3 hours. By reworking the CSS/Bootstrap of the home and instance pages, the team was able to make the website maintain a clean and pleasant look even on small windows/screens. This also means that the website maintains an organized look on mobile devices as well.

“As a developer, I want to be able to test my search and filtering functions so that I can ensure accurate results.”
This user story was completed by Sam, with an estimated time of 5 hours and an actual completion time of 2 hours. Because the team added three significant new features, we had to ensure that all functionalities were working as expected. This included testing corner cases, such as the user searching for an empty query.

DESIGN

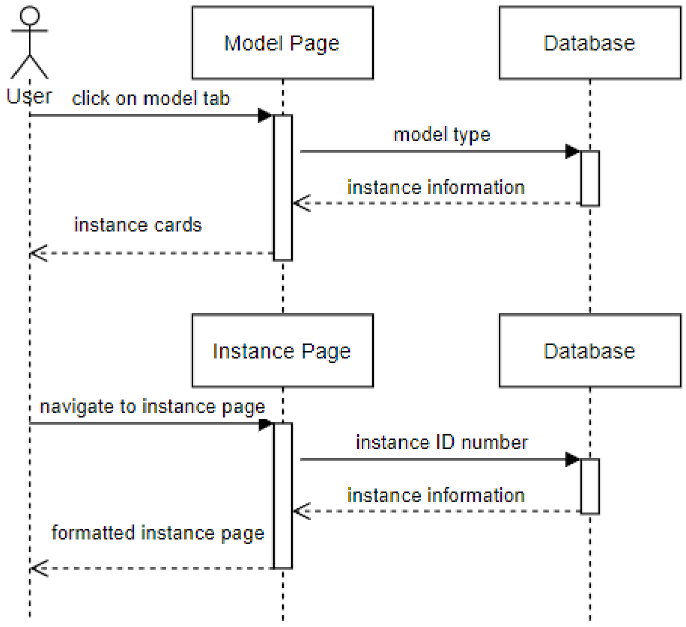
Our focus with the design of the website was easy navigation between information. Navigation on the website relies on links and the navbar at the top of each page. The navbar can lead you to categories of instances, allowing you to narrow your search. Meanwhile, each instance page also has links to other related instance pages. For example, a driver instance has a link to their current constructor. In general, the navigation use cases can be characterized by the following:



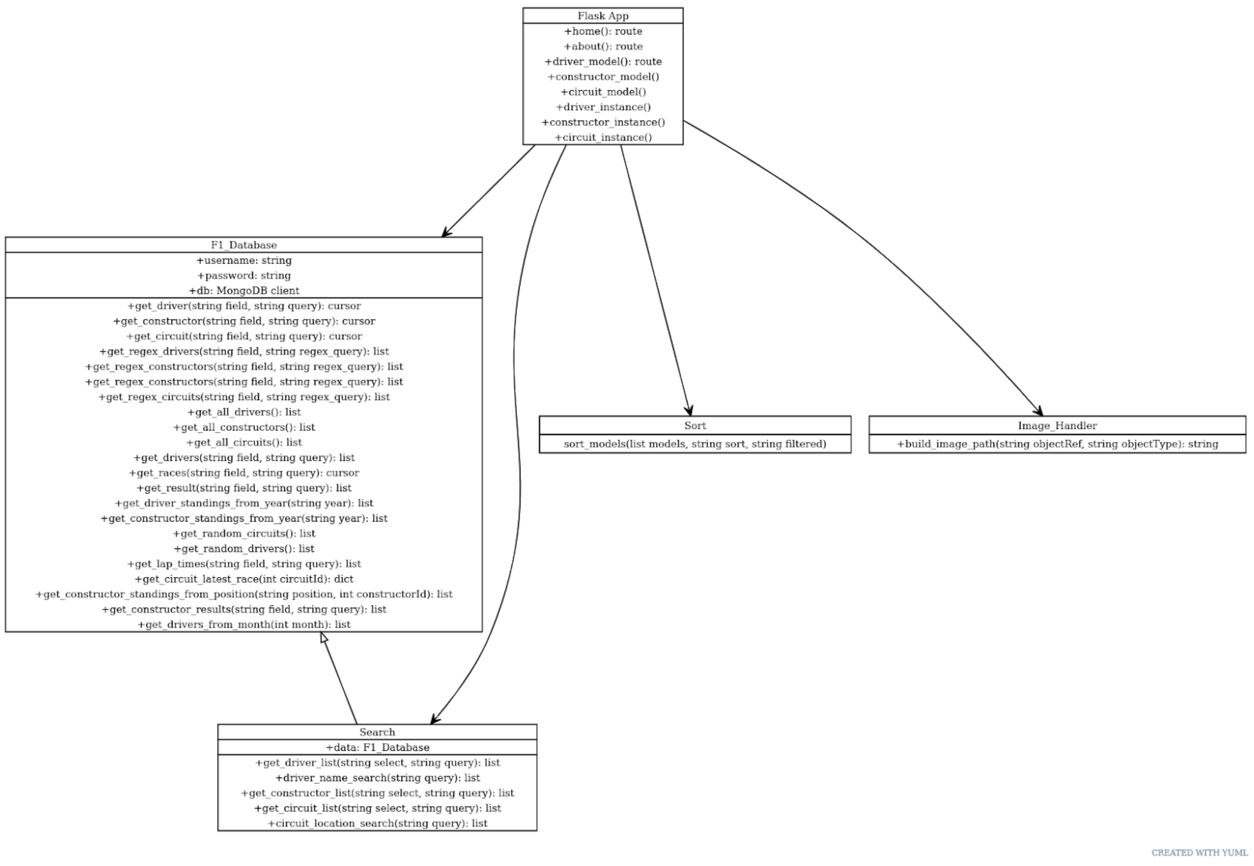
For current year: view upcoming races, driver standings, constructor standings
Includes
View Home Page
Includes
View driver's born this month, popular drivers, popular circuits,
Latest Races table and Victories table
Includes
View name, image, bio, DOB, nationality, latest constructors, past constructors
Includes
View Driver model page
Includes
View driver instance page
Actor
View Constructor model page
Includes
View constructor instance page
Includes
View name, image, bio, nationality, top driver (with points), latest drivers
Includes
Latest Race table and Latest Race Victories table
View Circuit model page
Includes
View circuit instance page
Includes
View name, image, bio, location (with coordinates)
Includes
View Fastest Lap of All Time table and Latest Race Result tables

This diagram guided the organization of our pages. We separated our site into 3 different categories of pages: home, model, and instance. The home page is the starting page of the site, with many links to relevant instances. The model pages separate the instances into categories of driver, constructor, and circuit, and allow the user to search and filter instances of each category. The instance pages then display specific information about each instance, with links to other relevant instances as well. Ease of navigation is maintained by a navbar at the top of each page, which links to our home, model, and about pages.

The use case diagram allowed the team to properly plan out everything we wanted to have included on the site. This was especially key on planning out what data should be featured on each instance page, as constructors, drivers, and circuits all have a similar layout, but different information. Once the content of the pages were established, the process by which this data would be brought to the user was planned with the sequence diagram shown below. This was to ensure that all data returns to the website and to the database were done in an efficient and straightforward manner.



During Phase 4, the team refactored much of the site's code to become more modular and implemented the model-view-controller design pattern (MVC). The data retrieved from the team's database would serve as the model, the Flask routing would control what specific data would be presented and sent between the model and view, and the HTML/Jinja templates would serve as the actual view and user facing side of the application. The refactoring was planned out by separating the project into those respective parts, as shown in the diagram below.


Flask App
+home(): route
+about(): route
+driver_model(): route
+constructor_model()
+circuit_model()
+driver_instance()
+constructor_instance()
+circuit_instance()
F1_Database
+username: string
+password: string
+db: MongoDB client
+get_driver(string field, string query): cursor
+get_constructor(string field, string query): cursor
+get_circuit(string field, string query): cursor
+get_regex_drivers(string field, string regex_query): list
+get_regex_constructors(string field, string regex_query): list
+get_regex_constructors(string field, string regex_query): list
+get_regex_circuits(string field, string regex_query): list
+get_all_drivers(): list
+get_all_constructors(): list
+get_all_circuits(): list
+get_drivers(string field, string query): list
+get_races(string field, string query): cursor
+get_result(string field, string query): list
+get_driver_standings_from_year(string year): list
+get_constructor_standings_from_year(string year): list
+get_random_circuits(): list
+get_random_drivers(): list
+get_lap_times(string field, string query): list
+get_circuit_latest_race(int circuitId): dict
+get_constructor_standings_from_position(string position, int constructorId): list
+get_constructor_results(string field, string query): list
+get_drivers_from_month(int month): list
Sort
sort_models(list models, string sort, string filtered)
Image_Handler
+build_image_path(string objectRef, string objectType): string
Search
+data: F1_Database
+get_driver_list(string select, string query): list
+driver_name_search(string query): list
+get_constructor_list(string select, string query): list
+get_circuit_list(string select, string query): list
+circuit_location_search(string query): list
CREATED WITH YUML

All functions for the site were previously found in the app.py file, but by modularizing the design, the team was able to achieve better organized code, information hiding, and easier adaptability for future added features.

TESTING

One of the most fundamental parts of any software development project is the testing and verification process. The team's website might not show glaring errors upon compilation, therefore it is necessary to carry out a series of tests of both the site's frontend and backend, to *better* verify that all functionalities and features are working as intended. This ensures both a smooth and frustration-free experience for the user, as well as bringing forward to the team's attention any errors or bugs that may have developed inconspicuously.

Phase 2

Phase 2 brought the website closer to its final look, and so the team began to conduct tests of both the frontend and backend to ensure all features of the site were working as expected. After learning about the Python Unittest library and Selenium in class, the team was able to utilize these tools to test the functionalities of both the user-facing website, as well as the database containing the project information.

Selenium GUI Testing

Selenium was used to test the webpage GUI. Tests were written for the navbar to ensure that the links sent users to the correct model pages. Furthermore, tests were written for the about page to ensure that they linked users to the GitHub repository webpage and to API homepages. Selenium was also used to test links on each instance page to ensure that they were properly linked to internal pages and to external resources. For each model page, there were 2 tests that we needed to do: testing pagination and testing whether the driver, constructor, and circuit model cards were linked to the correct instance page. We wrote 4 tests for pagination, testing the previous and next arrow buttons, as well as testing number buttons in between. For testing the correctness of the model card links, we compared the name of the instance on its model card to the name displayed on the instance page and ensured that they matched. All the driver, constructor, and circuit model pages had similar tests since they were structurally similar.

Backend Unit Tests

We tested the CRUD operations for each MongoDB collection that was used in the project. To test creation, we counted the number of documents before and after an insertion to make sure the total number of documents had been increased by 1. The Read operation test involved creating a document, pulling the document from the database, and checking to see if they were the same. The update test expanded upon the read test by editing the information that was retrieved and sending the data back to the database. We tested the delete command by deleting all of the items we created in previous checks and making sure the total number of documents has changed appropriately. We tested these operations because they are the four basic operations that are needed to operate a database. Our backend uses the read and update operations to gather and edit data in the database. The create and delete operations while not used by our website are used by the data gathering scripts to add articles to the collections.

Phase 3

Besides test refinement, the biggest addition in this phase that needed to be tested was the search, sort, and filter functionalities. Each model has their own unique filters that can be applied to their search/sort, meaning these had to be tested individually.

Searching, Filtering, and Sorting

Searching was implemented by sending regex queries to our MongoDB database. The *flask_pymongo* package allowed developers to easily query the database, and it even supported regex patterns and case-insensitive searches. Filtering was done by changing the field that was queried and sorting was done by applying sort lambdas to a list of database objects retrieved by searching and filtering. All of this functionality was provided to the user using drop-down forms and a search box. Users are also able to filter and sort blank with blank search queries to sort all instances available on the database.

Backend Unit Tests

The search, filter, and sort operations added a large testing requirement to our application. While we sorted the models using the same function, we implemented filtering and searching separately for each individual model. The reason we tested each function was to guarantee that our implementations worked correctly. To test search, I analyzed the functions and created 3 tests that examined various functionality in the functions. The first test uses a empty query to retrieve all the models from a collection. The other tests simply change the field that the search is looking in or change the number of tokens in a specific query. So, for instance, the test singleTokenCircuitLocationSearch searches for a circuit based on the location field using a singular token in the query. The sorting function was tested by inputting a list of models into the function and comparing the result to the original model list sorted by the python sort function. Finally, the remove accent function was tested because it allows the sort function to compare strings with unicode characters.

Selenium GUI Tests

Selenium GUI tests were expanded to cover searching, filtering, and sorting GUI elements on the driver, constructor, and circuit model pages. These tests select each option individually with and without pagination. These tests were used in conjunction with backend unit tests to test overall functionality of our app.

MODELS

The website is arranged according to three data models: drivers, constructors, and circuits. These models divide up the site into related modules, whose scope is broad enough to effectively organize all the Formula 1 data, while still being fine-grained enough to provide logical relevance to users.

Drivers

The driver model relates to Formula 1 drivers that are currently driving and those who have retired. With the website featuring over 830 drivers, this model is the largest of the three by a wide margin. Each instance page of a driver contains the driver's name, a short bio description, a driver picture, date of birth, nationality, driver number (only available for newer drivers), latest constructor, and past constructors. In addition to this information, two data tables containing the driver's latest race results and all of the driver's first-place finishes are also included.

The bio descriptions and pictures are scraped from the driver's Wikipedia page using the Python *BeautifulSoup4* library. A link to the driver's Wikipedia page is included at the end of the bio, should the user want to read more. The rest of the featured data points are provided by the team's aggregated project database.

Constructors

Constructors can be thought of as the "teams" that participate in Formula 1 races, with the term originating from the fact that teams are required to build the chassis that they use for competition. The site features 300 different constructors, including those which are no longer in operation. Each constructor instance page features the constructor name, a logo image, a short bio description, nationality, the constructor's top driver of all time (as well the points that driver earned while operating under that constructor), total race victories, and latest race victories.

The bio descriptions and pictures are scraped from the driver's Wikipedia page using the Python *BeautifulSoup4* library. The constructor's top driver is calculated by searching the project database for all races that the constructor participated in. The points earned for each race are then found, assigned to the appropriate driver, and the driver with the most total points is then returned.

Circuits

Circuits are the racetracks that the constructors and their drivers compete on. The circuits model also contains street circuits, which are public streets that are closed off in such a way as to create a racetrack. The website features more than 90 circuits which together are host to over 1000 races. Each circuit instance page features the circuit name, a picture of the track, a short description, the circuit location (and accompanying coordinates), the five fastest laps of all time to occur on the circuit, and the results of the latest race held on that circuit.

The descriptions and pictures of the circuits were similarly scraped from their Wikipedia page. The circuit's fastest laps of all time were calculated by finding all race results that occurred on a particular circuit and sorting by order of lap time. After the fastest laps were found, the accompanying driver, constructor, race name, and lap speed were also displayed alongside the lap time in a table format.

TOOLS, SOFTWARE, AND FRAMEWORKS

The team used a variety of tools and softwares to complete the project. These involved tools previously used by the team and those that were taught in class.

Flask, Python, and Jinja

The application code was written in *Flask* and Python, and the webpages were written in HTML with Bootstrap components. Jinja templating was also used to help programmatically generate some webpage features at runtime. Initially the site was very simple, with hardcoded HTML values and only a few webpages. As the phases continued, the team was able to use these tools to create a vast number of webpages with HTML and Jinja components that allowed for a more interesting site.

MongoDB Atlas and PyMongo

MongoDB is a cross-platform document-oriented NoSQL database program that was used to house all the data for the website. The team used a system of integer IDs for drivers, circuits, constructors in order to easily reference the models between each other. Since *Flask* was used as the application framework, the *flask_pymongo* package was used to allow the application to access the MongoDB database and *pymongo* was used to access the db outside of *Flask*. The team initially populated the database with the information found in the various data sources. As the project continued, the team's attempt to extrapolate certain data elements (such as a constructor's top driver) resulted in very slow webpage load times. This was alleviated by calculating these values in separate scripts and then populating the database with the new values, which can then be quickly referenced later in the site.

Bootstrap 4

The team used Bootstrap 4, the open-source CSS framework, to make the site responsive. Bootstrap is a powerful framework that the team did not fully use until later in the phases. Once more practice was had with the various components, including responsive tables, images, containers, and columns, the team was able to create an aesthetically pleasing site layout that maintained its look regardless of screen size.

Selenium and Python unittest

Python's *unittest* module was used to test application code, while Selenium was used to test the website GUI. The unittest modules particularly focused on CRUD database operations: create, read, update, and delete. Selenium ensured that all the user interactive components of the site were working as intended. The tests were relatively simple at the start of the project, but were later refined to become more intricate and to ensure more coverage. Such tests are discussed more in-depth in the testing section of the report.

Additional Libraries

Additional third-party Python libraries were used with Python and its standard library to perform various functions and to collect data for the application. The *requests* library was used to harvest data by sending GET requests to the API services and *ratelimit* was used to pause API requests on request failure without terminating the program. The *bs4* (Beautiful Soup 4) package was used to scrape images and biographies from websites as this information was not available from any API services. In addition, *PIL* (Python Imaging Library) was used to convert all collected images into a uniform format.

SOURCES

Although the team began the project with a variety of knowledge and industry/academic experience, there were still many things that we had to learn in order to create the necessary features for the website. For this, many sources outside of class were consulted.

[1] Luvsandorj, Zolzaya. “Two Simple Ways to Scrape Text from Wikipedia in Python.” *Medium*, Level Up Coding, 19 July 2020, levelup.gitconnected.com/two-simple-ways-to-scrape-text-from-wikipedia-in-python-9ce07426579b.

This tutorial was used to learn how to scrape information from Wikipedia for the bio/about section found in each of the model instance pages. The article goes through the process of using both the Wikipedia Python library and urllib & Beautiful Soup (bs4). The latter method was used for the website, as the team already had the Wikipedia URL for each driver, circuit, and constructor (from Ergast). The Wikipedia library’s search function only takes a search term as an input, not a URL, which commonly lead to incorrect articles being found.

[2] Mwiti, Derrick. *Introduction to MongoDB and Python*, Data Camp, 30 Oct. 2018, www.datacamp.com/community/tutorials/introduction-mongodb-python.

The tutorial above was as a reference for our mongodb atlas instance. We followed the tutorial to set up our cloud mongoDB. After scraping the data from the database, we programmatically added the data to the database using the methods described in the tutorials. The tutorial also provides code for retrieving elements from a collection in the database.

[3] “FlexBox Exercise #4 - Same Height Cards.” CodePen, n.d. codepen.io/veronicadev/pen/WJyOwG.

This code was used as a template for building the instance cards on the model pages. We chose to use this as a template because it was an aesthetically pleasing way of displaying relevant instance information. This template has helped us to refine the look and functionality of our website.

[4] bakkalbakkal. “Simple URL GET/POST Function in Python.” *Stack Overflow*, 9 Apr. 2012, stackoverflow.com/questions/4476373/simple-url-get-post-function-in-python/4476389.

The Stack Overflow thread cited above provided code that helped the team understand how to send GET requests to retrieve information from an API service. Furthermore, it demonstrated how python can parse JSON responses.

[5] gyoho. “Python API Rate Limiting - How to Limit API Calls Globally” *Stack Overflow*, 24 Feb. 2017, stackoverflow.com/questions/40748687/python-api-rate-limiting-how-to-limit-api-calls-globally.

This Stack Overflow thread helped team members understand how the *ratelimit* package could be used to time out a program when an API rejects the request. It also demonstrates alternative ways to limit API requests without *ratelimit*.

[6] Martijn Pieters. “Getting a particular image from Wikipedia with BeautifulSoup.” *Stack Overflow*, 18 Jan. 2015, stackoverflow.com/questions/28006690/getting-a-particular-image-from-wikipedia-with-beautifulsoup.

The Stack Overflow thread provided instruction on how to filter out unwanted images while scraping a website like wikipedia for images. This method was able to filter out most images, but manual image review was still required.

[7] dm2013. “Convert png to jpeg using Pillow.” *Stack Overflow*, 6 Apr. 2017, stackoverflow.com/questions/43258461/convert-png-to-jpeg-using-pillow.

This stack overflow provided insight on how to use *PIL* (Python Imaging Library) to convert from images using png format to jpeg format. This was used as a basis to write the script that converted all images from jpeg format to png format.

REFLECTIONS

After each phase, the team took time to reflect on what we learned, things we struggled with, things that we need to improve, and things that worked well. By taking time to analyze our past work, the team can steadily improve and adapt to ensure a smooth development process.

Phase 1

In phase 1, the team created a simplified, static version of the website. While the team had completed tutorials on using Bootstrap and Jinja templates in class, this phase saw us learn how to use these tools in much more effective ways. Bootstrap was used to display multiple objects on the page in such a manner as resizing the window/screen allowed for the components to dynamically adjust themselves. The team used Jinja to create more complex templates for the models and the model instance pages.

During the development process, there were certain obstacles that the team struggled with along the path to completing the phase requirements. These struggles include learning the more complex Bootstrap components (e.g. keeping Bootstrap cards not only the same width, but also the same height), writing the initial scrap for harvesting the data from our APIs, getting the web page CSS to display our components in the desired way, learning how to use more complex Jinja components, and integrating Github within our Slack.

Aside from technical difficulties, there were also some things that the team realized we have to improve on for future phases. Namely, the team needs to learn to communicate more often, respond in a quicker manner, and as a whole, begin work on the project earlier.

Nevertheless, there were many things that the team did that ended up working very well. The team will keep this in mind as the project development continues. These things include the utilization of Slack for team communication/link compilation, using the Github issue tracker to ensure all requirements are met, using Github user stories to easily create functionalities and implement features, hosting the twice-weekly meetings on Zoom, and using the tools taught in class (Flask, Bootstrap, and Jinja) to create the website.

Phase 2

In phase 2, the team learned how to collect API data, scrape images, connect a database to flask app, edit database artifacts, add dynamic content to the website, test GUI’s, and test application code. Developers were able to further explore material presented in class and in tutorials which covered the aforementioned topics during this phase. The “requests” python package was used to collect information from the API, and the “BeautifulSoup4” python package was used to scrape for driver, constructor, and circuit images. MongoDB was selected as the project database since its NoSQL structure allowed the team to change database fields without redesigning the entire schema. Selenium was used to test the website GUI and the *unittest* python package was used to test application code. The team familiarized themselves with GitHub issue assignment helped developers track project progress.

The development process went smoothly overall, but the team struggled with several issues during this phase. The first issue encountered by the team were API timeouts from frequent requests, which was solved by implementing a cooldown timer after an API request failure. Next, several team members found that making numerous database calls upon loading the page was extremely inefficient. This issue was addressed by performing analysis of the data ahead of time and storing the results on the database. The team also had issues with scraping numerous invalid images for thousands of instances, and spent many hours pruning the image set to only include relevant images. Another issue that hindered the team was inflexible HTML pages that relied heavily on Jinja templating. This issue was resolved by rewriting the problematic pages to rely more on passed parameters. Finally, the team found that there were null artifacts that remained from collecting API data. The team is currently deliberating on the best solution for this issue.

Although the team faced several issues that set back development, other things went smoothly. GitHub issue tracking, issue assignment, and project boards were crucial in keeping the project organized. Furthermore, local git branches allowed developers to work on different features without constantly running into merge conflicts. MongoDB's NoSQL structure proved to be very convenient since it allowed developers to easily add new classifications of data. Furthermore, MongoDB's Cloud Atlas GUI was also very helpful since it allowed a developer to check that their database modifications were successful. Finally, open-source python packages such as "requests" and "BeautifulSoup4" made complex tasks like collecting API data and scraping images extremely simple.

Phase 3

In phase 3, the team learned how to search a MongoDB database using Flask, how to convert non-ASCII characters to ASCII, how to properly use Bootstrap for mobile/small screen displays, and how to pass data to the backend with forms. While the team had used MongoDB queries fairly often up until this phase (i.e. db.collection.find()), the team now learned how to use the db.collection.search function combined with regular expressions to efficiently search the database for string content. Because some of the model names feature non-ASCII characters, such as François Hesnault, the team had to learn how to convert these into workable ASCII characters. Although Bootstrap had been used for the website throughout development, to get a the webpage displaying properly on small screen/mobile devices, we had to learn more about the specific mechanics of Bootstrap and proper layout philosophy. To implement the search, sort, and filter functionalities, the team had to create an HTML form to allow the user to send their own chosen attributes to the team's appropriate functions.

The team had established effective methods of communication and work tracking, leaving most challenges to be technical in nature. The first struggle involved the way driver names were stored in the database, being split into a forename and surname field. This meant we were not able to search one query against all fields. This issue was resolved by searching for individual tokens across all possible fields. Similarly, circuit location queries faced a challenge with their location and country being split into separate fields. This was resolved in the same way. Accent characters found in some driver and circuit names caused some instances to appear at the end of our searches since accented characters have higher unicode values than ASCII. We fixed this by converting all characters to ASCII in our search lambdas.

Despite these technical hurdles, a number of things also went well for the team. The MongoDB API provided by *flask_pymongo* was extremely helpful and it was really easy to make case insensitive searches with regex. The use of detailed comments, including function comments describing expected inputs and outputs, helped developers understand the requirements for the functions they wrote.

Phase 4

In phase 4, the team learned how design is a failure based process. Failure is an important part of the design process because it reveals the parts of a project that do not work or can be improved upon. This was clearly evident as we refactored our code base, removing dead or duplicated code. As we implemented the MVC design pattern, our code became more organized and modular, further demonstrating the importance of proper design. Refactoring allowed us to re-evaluate the failures in our code, and improve upon them.

This phase, the team faced several challenges, both technical and non-technical in nature. Because of Thanksgiving break, our normal meeting schedule was interrupted, and we were unable to have live meetings. Our individual schedules also shifted, and it was difficult to work throughout the break. We had some difficulties refactoring our original code base because it had been written without regard to modularization. Much of our time was spent organizing our code with a more modular structure. The team also had difficulties both choosing and applying a design pattern that was applicable to our application. This was also due to the difficulties we experienced with refactoring. A clear design pattern did not arise until the code base had become more organized.

Although the team faced several difficulties throughout this phase, many things still worked well. Slack has been a consistently effective means of communication, even with the break. The GitHub issue tracker has also kept the team on track and aligned with the tasks that need to be completed. Continued usage of detailed comments and commit messages has helped the team organize the project, allowing us to easily review our code and make changes. The MVC design pattern also worked very well with this database application. Python allowed the team to quickly refactor code, streamlining our refactoring process.

EE 461L Team Compact (Team E2)

When:

Mon, Wed 2-4 pm

How to make decisions:
Decisions will be made with a vote that must be ¾ pass. In the case of 2/2, flip a coin and move on.

How to divide work and set deadlines:
We will volunteer for work, and then distribute the remaining equally based on experience. We will vote for a leader. The leader will set project deadlines, with a minimum of 2 days before the actual course deadline. This is to allow us to have adequate time to review all work before submission.

Punishments:
If you are late on a deadline, you get the last pick for work next week and you have to write an increasing amount of the written reports. Plus you must apologize for your incompetence in the next Zoom meeting.

Resolving Conflicts:
In the case of a conflict, members should first attempt to resolve it through respectful discourse. A team member should never, for any reason, disrespect or criticize another member. If a conflict cannot be resolved through a respectful discussion, a neutral third party (another team member or, if necessary, Prof. Eberlein/a TA) will be consulted as an arbiter.

Communication:
The Slack workspace will be the main form of communication. Messages can be sent as often as needed, and while teammates may be busy at times, they should at least respond with an acknowledgment that they have seen the message. If teammates will be unavailable for contact for an extended period of time (e.g. broken phone), they should notify the group ahead of time or notify the group with a Canvas message.

Code Sharing:
Code should be kept and updated through the team Github to ensure consistency. All issues will be tracked on GitHub.

Team Relative Strengths:

Edie: Python, CSS, HTML

Sam: Javascript, Typescript, Angular, Python, CSS, HTML

Alan: Python, Javascript, CSS, HTML

Kevin: Python, Javascript