The Drag - Phase II

TECHNICAL REPORT

GitHub Repo: github.com/53Dude/TheDrag.git
Site Link: thedrag.appspot.com

Daniel Lazcano
@: daniellazcanoplus@gmail.com **Git:** github.com/DanielLazcano

Ethan Santoni-Colvin
@: ethan.santonicolvin@gmail.com **Git:** https://github.com/ethansantonicolvin

Frank Le
@: fle734449@gmail.com **Git:** https://github.com/fle734449

Guy Sexton
@: gsexton@utexas.edu **Git:** https://github.com/53Dude

Jonathan Walsh
@: jdwalsh79@gmail.com **Git:** https://github.com/jdw4867

Kishan Dayananda
@: kishdaya@gmail.com **Git:** https://github.com/kish314

Phase I Lead: Kishan Dayananda
Phase II Lead: Ethan Santoni-Colvin

Group: Morning - 4 / Pit Crew
Slip Days Used for Phase II: 1 day

Motivation and Users

TheDrag is a website built for users who want a consolidated place that holds all the automobile shopping information they would need. Whether it's a college student looking prices and dealerships for their first car purchase, speed-demon car enthusiasts looking to find the next hot rod on the market, or a conserved environmentalist who needs to get to where they but wants to find the least carbon emissive vehicle that they can get; TheDrag serves to bring all that info right into your web-surfing device.

User Stories

Our Own:

1. I hate really long pages of lists and really short pages of lists on websites. I wish there was an option like on other sites to go through pages to see a certain amounts of things.
    a. Estimate: 4 Hours
    b. Actual: 7 Hours
    - User story essentially describing a need for pagination so that a single page doesn't get too littered with elements/info that can also appear overwhelming to the average user.
    - The estimated 4 hours was due to finding existing bootstrap JS support online and thinking that we only had to implement said support. However, we ran into issues with pulling JSONs and general data into said JS files. So we scrapped that and figured out how to implement it in JSP, increasing the estimated time to about 7 to 8 hours.
2. As a car fanatic, I can easily get lost browsing these amazing car websites, like TheDrag! I need a way to get back to the home page.
    a. Estimate: 2 Hours
    b. Actual: 1 Hour
    - We edited the navbar on the JSP files so that the website logo included an HREF that takes the user back to the home HTML page. Tricky part was editing the navbar but turned out to take less time than estimated.
3. As a first time car buyer and opportunist, I am very concerned about missing a good deal. I would love a website that shows all the cars I'm interested in within my area and updates that list too!
    a. Estimate: 8 Hours
    b. Actual: 15 Hours
    - User story involves showing many listings in the Austin area. So we made a web-scraper to search through listings in the area to pull data and compile a database of listed cars. This list can be easily updated by running the scraper. The pricing data is also acquired through the scraper and then displayed on the car listing page.
4. As a user with age-related macular degeneration, seeing blurry or disproportionately sized images on websites puts a strain on my eyes. I would love some uniformity in image styling across a website!
    a. Estimate: 3 Hours
    b. Actual: 4 Hours

- This user story is describing an improvement from last phase in terms of visual continuity that enhances the user experiences. Mainly involved some Adobe dreamweaver and string manipulation within the JSP files. The actual time for the general HTML/CSS styling was longer. (Other than the image styling as described in this user story).

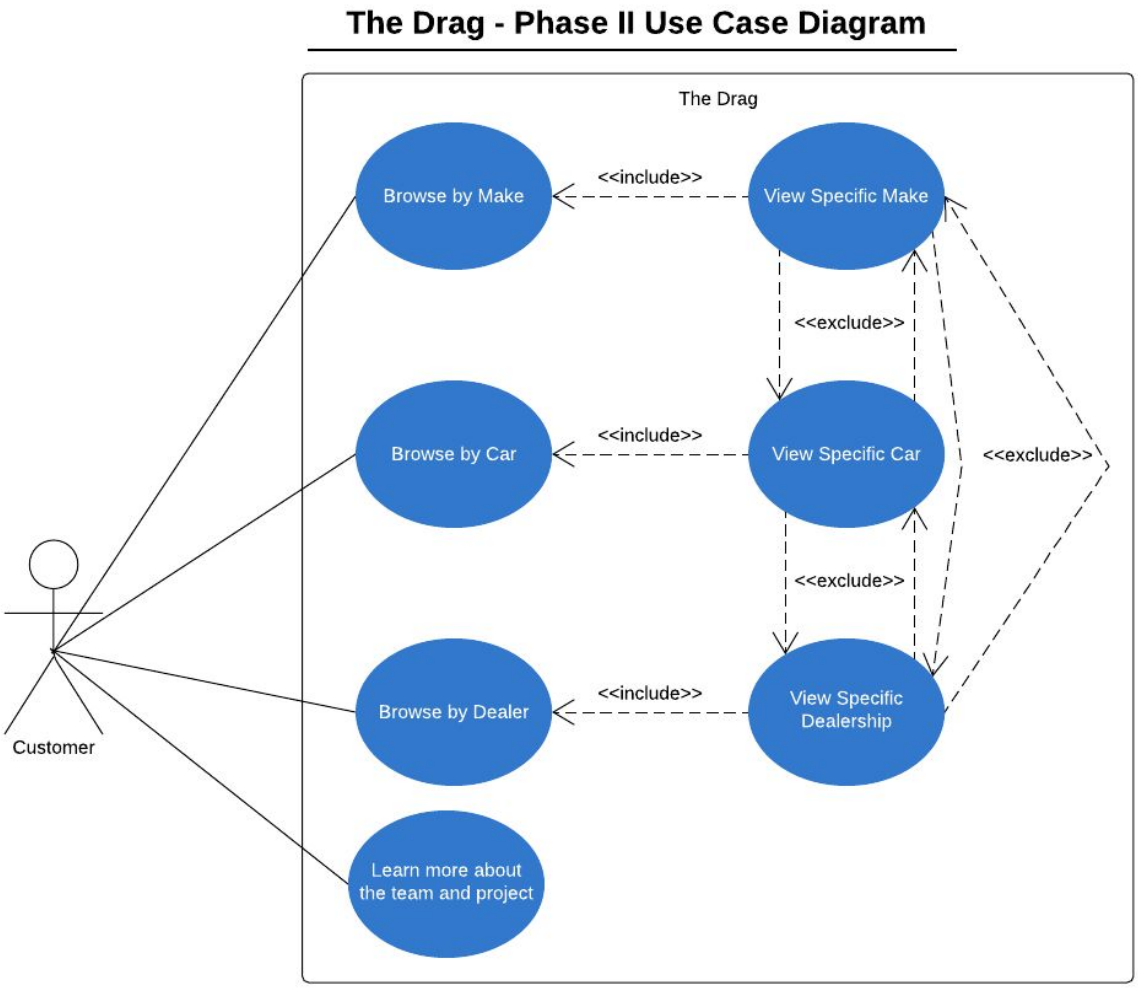
5. As a commuter to work, I want to know the highway mpg of certain models.
    a. Estimate: 3 Hours
    b. Actual: 3 Hours
    - This user story described an original idea/want on the conception of TheDrag. This involved a back-end member showing a front-end member how to do API calls from the NHSTA. Involved making servlets and making helper functions and instantiating them for every JSP model file.
6. I want a list of current manufacturers that are sold around my area.
    a. Estimate: 2 Hours
    b. Actual: 4 Hours
    - Involved web scraping with the backend team and passing it to the front end team to dynamically create the instance pages in the model listing JSP files.
7. I'm a die hard Subaru fan, I need to know which models are made by Subaru.
    a. Estimate: 5 Hours
    b. Actual: 3 Hours
    - The data for this was already acquired in the data scraping and implementation to actually render and display the data was very simple after the completion of the dealer and car instance pages.
8. As a undergraduate college student with no income, I want a way to find a cheap car under $20,000, so I can drive to school.
    a. Estimate: 3 Hours
    b. Actual: 3 Hours
    - This user story expresses a need to list an attribute of each of the cars that are listed as being available for sale. Its an essential part of the service we want to provide to the user; namely that they would be able to search for cars in their price range.
9. As a car enthusiast I want to know what exact body type of the car I'm interested in.
    a. Estimate: 3 Hours
    b. Actual: 3 Hours
    - This user story describes one of the main audiences that TheDrag is catering too and implementation is similar to user story 5 where API calls are made from NHSTA.

10. I'm in the market for a car. More than just knowing the price, I want to know what the car will look like before I buy it.
    a. Estimate: 2 Hours
    b. Actual:
    - Beyond knowing text attributes describing the car, a user would want to know what the car would look like to be interested in buying it, and thus this is a key user story to satisfy. Including images of the cars also enhances the aesthetics of the site.

Customer Stories:

1. As a concerned user of who is making my product I would like to see the developer team of the product.
    a. Estimate: 2 Hours
    b. Actual: 1 Hour
    - Created about page that shows developer team
2. I want to be clear about the purpose of the website.
    a. Estimate: 1 Hours
    b. Actual: .5 Hours
    - Landing page shows purpose of the site.
3. As a user with bad eyesight I want to be able to use the website easily for someone with my disability.
    a. Estimate: 2 Hours
    b. Actual: 2 Hours
    - We located a site-wide theme that is easy to read and cleanly displays all the info on each page.
4. As someone who browses the internet everyday, I would like a clean and polished UI.
    a. Estimate: 3 Hours
    b. Actual: 3 Hours
    - Describes a want from a user to not view a visually unappealing website.
5. I would like to know the closest locations of dealerships selling a certain make.
    a. Estimate: 3 Hours
    b. Actual: User describes the functionality the website can give to them.

Use Case Diagram



This UML Use Case diagram shows how users will interact with our site and browse through all of the three models. The User can initially select one of four options. Three of them will give the user a listing of instances to pick. After picking an instance they can learn more about it or use page links to see other related instances.

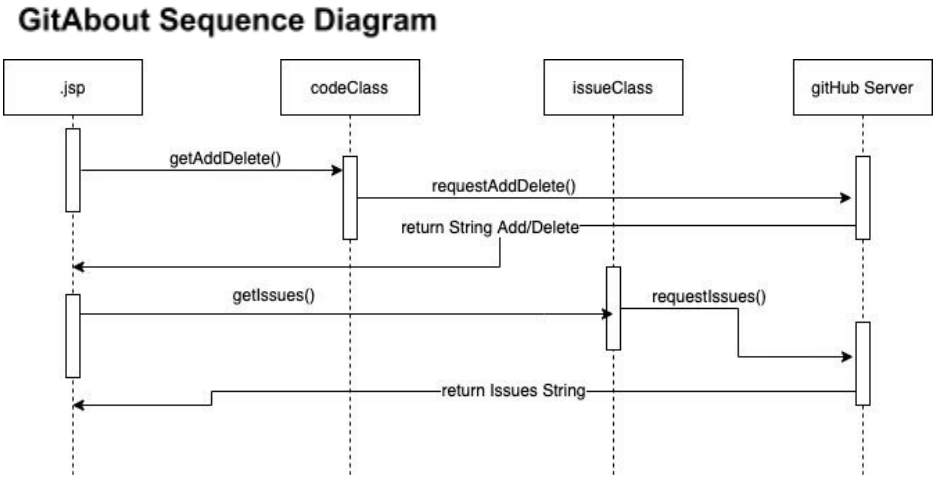
Design

Our design was essentially giving users the options of choosing to browse by make, specific car, and dealership. Clicking “browse by make” will take you to a list of makes available. Clicking a specific make shows a list of car instances that are of that make to click. Clicking a specific car on that page takes you to information about that car such as an image, price, mpg, horsepower, and the dealership that possesses this car, among

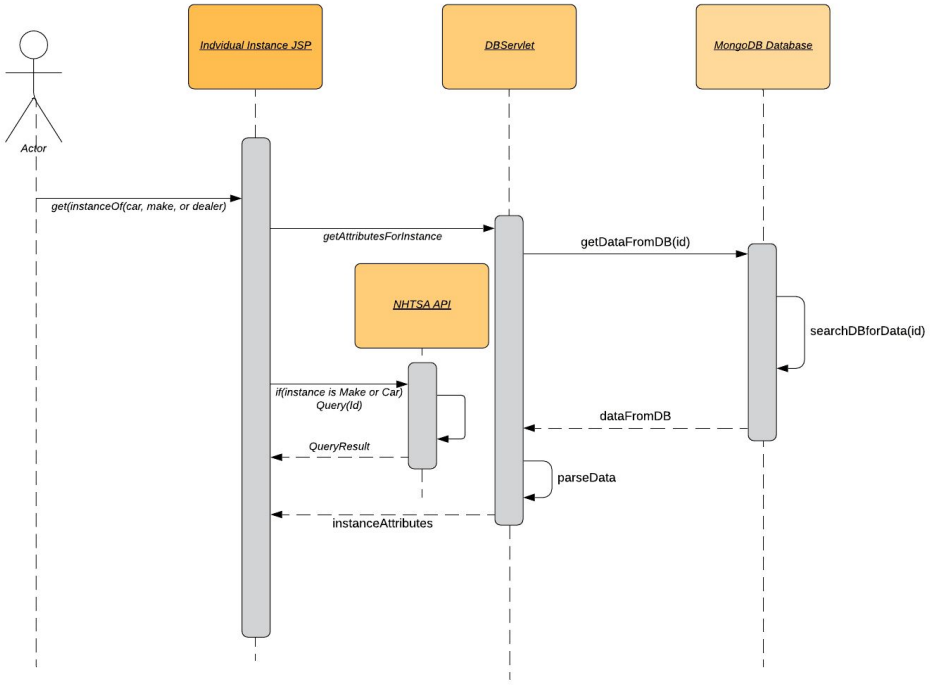
other things. One can access this list of cars directly by first clicking on “browse by car” at the top on the navbar. Clicking “browse by dealerships” up top takes you to a page listing dealerships nearby. Clicking a single instance of these takes you to a detailed description of the dealership, along with an image and address. It also reveals all the available cars that are sold at that dealership. So basically, make, model and dealership are all linked to each other in some way where you can access any instance from another instance.

We derived our About page statistics dynamically from GitHub by making calls to two different Java classes in our jsp file, CodeStats.java and IssueStats.java. CodeStats pulls GitHub’s information on the number of commits, while IssueStats retrieves the number of issues assigned from our GitHub. If the request to the GitHub API is denied, it will print an error message and still show the about page, otherwise it will print the information. If GitHub statistics doesn’t show up, wait for a while and retry, this issue is due to the GitHub API, not our code (per TA). Also note in order to get to the about page from the home page you need to press “about” at the footer of the webpage.

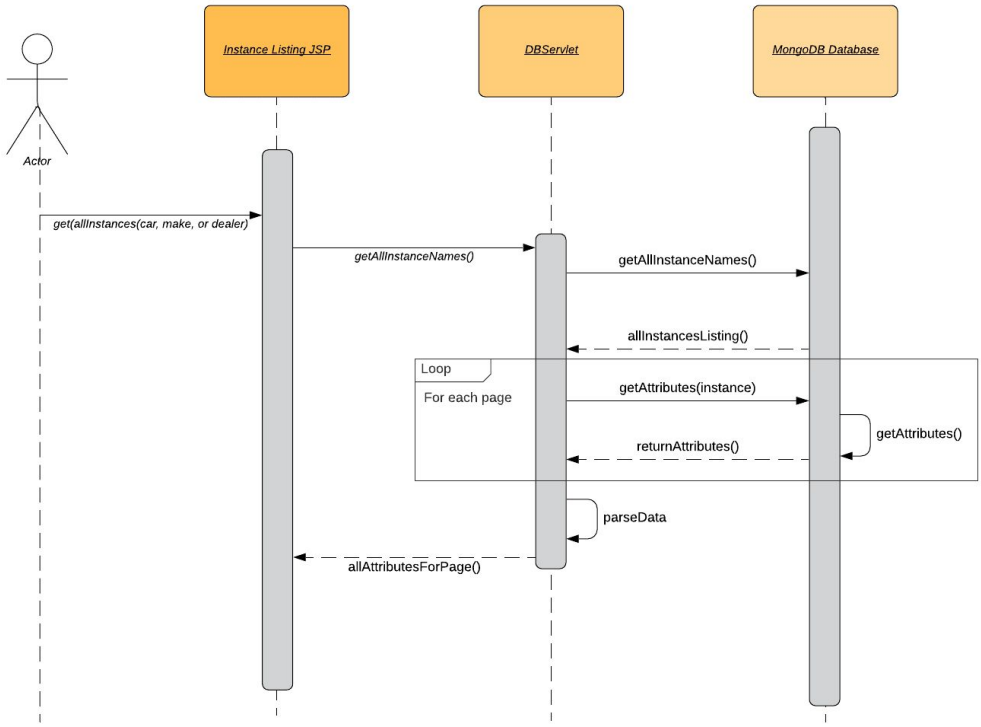
Responding to a comment from Phase I: The site logo functions as a home button, following the convention from most other websites. There is also a secondary “Home” link at the bottom of every page as a backup.



Instance Sequence Diagram



Instance Listing Sequence Diagram



Pagination

Pagination was fun to figure out. The front-end team initially created pagination using JavaScript. We were initially using a local JS array to implement this with the intent to pull from the database into that array. This was not possible as our database was not setup to use Node.js. We then recreated the pagination in a JSP, using Java servlet code to generate the pages. We utilized some CSS tricks in the elements to make all the listings render in a grid and then used the length of the arrays to dynamically generate all the needed page numbers. We have a jump to first, jump to last, previous, and next button along with the 5 closest pages numbers. This limiting ensures that there are not too many buttons displayed on the page at once.

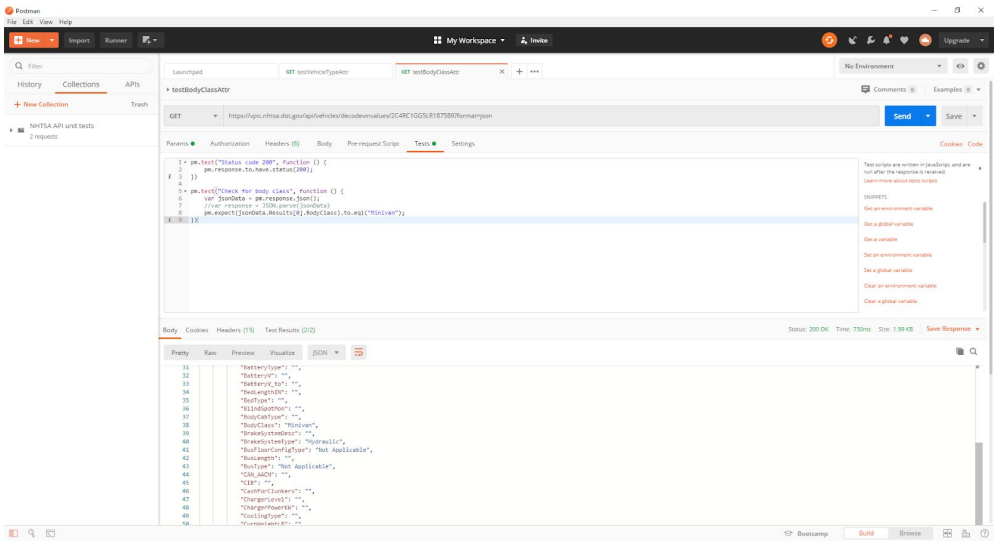
Testing

All unit test files can be found in src/test

Backend: Total - 21

Postman used to test API calls:

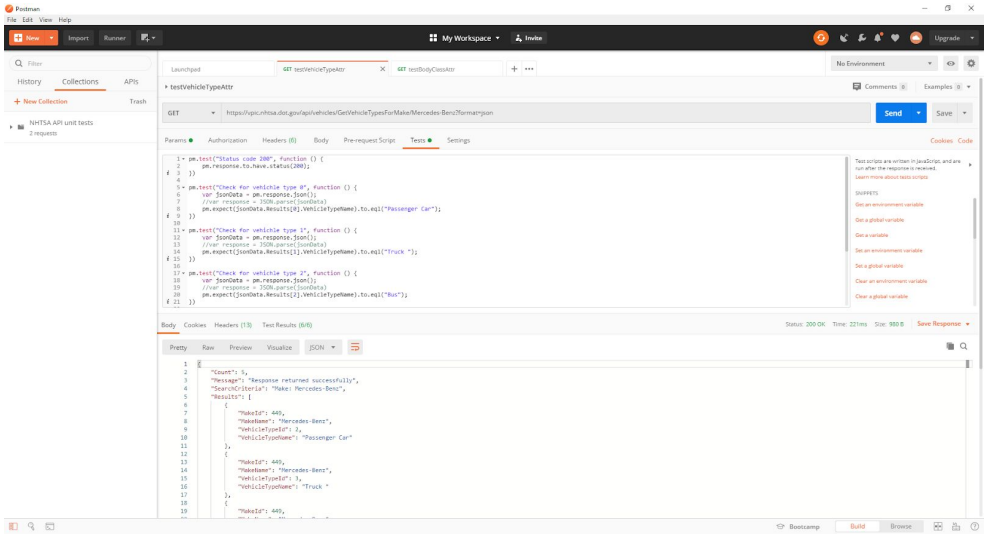
testBodyClassAttr



Description:

This test is used to check if we can successfully call the NHTSA API and to check if we can get the body class attribute of a car instance. Each car instance has a unique vin which we can use to make the call to the api for that specific car's attributes. In our case we already had two other attributes scraped for car instances using our web crawler/scraper, so we decided to use the NHTSA API to get a car's body class.

testVehicleTypeAttr



Description:
This test is used to check if we can successfully call the NHTSA API and to check if we can get the different vehicle types attribute of a make instance. Each make instance is identified by its name which we can use to make the call to the api for that specific make's vehicle types. In our case we already had two other attributes scraped for make instances using our web crawler/scraper, so we decided to use the NHTSA API to get the different vehicle types of a make.

JUNIT used to test web crawler/scraper:
ScraperTest.java
Description:
This test is the bulk of our backend testing because it tests our web scraper/crawler function which scrapes information for all three of our models and their instances. It first tests to see if it scrapes the makeIDs off of cars.com correctly. Next, it tests to see if the scraper successfully scrapes all makes and their respective logos off of carlogos.org correctly. The next section of tests, scrapes dealership information and checks that all fields are stored correctly. The remaining tests check the correctness of collecting data on our car instances. The first few tests of this section cover scraping for the fields of a car object. Finally the last part of the test involves checking the interconnectedness of our models. We store each model as a HashMap that can be traversed to find specific instances. Each model has an array for the other models inside each of its instances, which stores their unique identifier. The test checks to see if the data is stored correctly for all three models and then checks to see if they all have a has-a relationship with each other.

Testing while running the code with print statements:

When writing our web scraper, we also wrote print statements to check the progress of our scraper when it was running. We also printed out the HashMaps for car, make, and dealership to ensure they were getting the correct information from the website we were scraping. As we continually wrote the code, we added print statements to check every attribute we scraped from the website was correct. We did this because a full scrape would take a little bit over an hour and a half, and we wanted our scraper to collect the right DOM elements before making a full run.

JUNIT Front-End Selenium Testing:
To test the front-end/GUI portions of our website we utilized Selenium IDE testing. This allowed us to download a developer version of Selenium as a Google Chrome extension. This allowed us to run TheDrag on the local host and on the deployed appspot while recording “clicks”/user GUI inputs and convert them, within the Selenium IDE, to code syntax. We mainly tested the interconnectivity of the different model listings/instances, navigation bar, GUI buttons, and elements. Selenium IDE then allowed us to export the syntax into a JUNIT java file coupled with modifications allowed us to run an automated FireFox browser that ran the code lines. We had to include system waits to allow the browser to properly load due to the high amount of data being pulled from MongoDB and NHTSA API calls. If it wasn’t for this portion then the test would run into errors due to elements of the next line of the recorded tests not appearing in the current HTML document.

JS Mocha Tests:
As all of the dynamic front-end element/GUI generation are in JSP files rather than JS files, the Mocha tests were replaced with Selenium JUNIT tests that achieved the same functionality. We confirmed this particular issue with our TA, Ashwin.

Models

Our three models are the make of a car, the unique car itself, and the dealerships. When accessing makes of car, you have options to choose from that then take you to a bunch of different cars of that make. You can also access these cars by clicking browse by car as well and clicking browse by dealerships will take you to a list of dealerships, each linked to an instance of a specific car. We scraped data for makes, car, and dealerships from various APIs and websites. These sources are listed below:

Source 1: https://www.cars.com/
Source 2: https://vpic.nhtsa.dot.gov/api/

Source 3: https://www.carlogos.org/

Unique cars are scraped by our custom Cars.com web scraper. We scrape all new cars available for sale within a 20 mile radius of campus, specifically grabbing their VIN, price, associated dealership, and mpg. The VIN of each car is fed into the NHTSA API to gather additional car information such as the body type and engine horsepower. Altogether we have over 14000 cars on our site.

Makes are generated by bucketing the data scraped from Cars.com by make. Images for each make are gathered from programmatically carlogos.org. Dealerships are also created via bucketing the individual cars.

The models are deeply linked. For example, if you click the button to browse by make to start, you will be presented a list of cars (another model) that are of that make. Then clicking on one of those cars, the car's unique, dynamically-generated page presents options to see all cars sold by the same dealership (another model) that sells that specific car, as well as to go back to seeing all cars of that make. Similar linkages occur should you start by browsing by dealership or car.

Tools, Software, Frameworks

The Pit Crew (The development team behind TheDrag) utilized the following technical tools to complete Phase II:

Eclipse - The main IDE everyone used to code the Java, CSS, HTML, JSP, XML, etc. files and to run the website on the localhost through Google App Engine integration.

Google AppEngine - The PaaS used to host and do webby stuff.

Adobe DreamWeaver - The Front-End team used this visual to code editor for streamlined editing/development of Bootstrap, CSS, and HTML files.

Bootstrap - A bootstrap design was found online and implemented into our front-end web design.

Maven - Used as our project builder and to quickly add third party libraries for us to use in our project

OKHTTP by Square - A request-response API that allows us to collect data on our instances through making a request to other APIs.

JSOUP - An open source Java HTMLparser library that we used to scrape websites. We used this tool to scrape two of our data sources, cars.com and carlogos.org for the information we needed for our models.

Jackson - A java based library we used to map our java objects (data we scraped for our models) to JSON. After our web scraper finished scraping the websites and stored the data into objects, we mapped them to three separate JSON files which we would later store into our database.

JUNIT - Unit testing framework for java. We used this to test our java code, specifically our web scraper.

Postman - API used to create collections and test API calls. We used this to test our RESTful API and getting data from APIs such as NHTSA.

MongoDB - Our database program. We use this to store information about the cars, dealerships, and makes listed on our site. Our cluster is built on Atlas, MongoDB's free cloud storage.

Selenium IDE - Our front-end/GUI automated testing framework. We used its chrome extension version and exported JUNIT test cases as talked about in the above testing portion of this report.

Reflection

Our team had good communication going throughout phase II via our Slack channel. We all made significant efforts to meet up whenever there was a lot of work to be done and were good at letting everyone know when one of us couldn't make it for some reason. We were tremendous at pair programming and it made it much easier to troubleshoot. Front-end and back-end teams worked very well with each other to make sure that everyone was on the same so errors would be minimized in the long run. The team just had great synergy.

Some struggles our team faced were some members having subscriptions to more advanced software to get front-end work done, which is also associated with differences in expertise. But we got around that with free trials and student memberships. Other struggles included our schedules not lining up at times and some people would inevitably have to be left out. We did a good job of tuning in remotely if busy, through hangouts or facetime, but sometimes it was just unavoidable.

We learned what it was like to work in a more team-dependent environment where roles had to be delegated and trust had to be developed for others to get their work done. We learned the necessity of JSPs despite our hatred towards them. We better understood servlets and the calls necessary to make them work, and how to encode JSON files to gather data from them.

Looking forward to Phase III, some of the considerations we will revisit are expanding the radius from which we scraped our info to beyond 20 miles, parsing out dealerships that had no cars to sell, and further cleaning up url slugs and UI/UX.