The Drag - Phase III

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
Phase III Lead: Guy Sexton

Group: Morning - 4 / Pit Crew
Slip Days Used for Phase II: 1 day
Slip Days Used for Phase III: 0

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

Developer User Stories:

PHASE III

1. I’m browsing the dealers in my area and I would like to know if any dealerships specialize in selling Honda cars.
    a. Estimate: 2 hours
    b. Actual: 30 mins
    - This was implemented by filtering each dealer on the listing pages for the specific make(s) of cars they specialize in selling.
2. Tom suffers from an auto-immune disease. With the current stay-at-home order due to COVID-19, Tom needs a way to safely browse the cars available in his area without waiting on the phone for a salesperson to walk through each car for sale.
    a. Estimate: 1 hour
    b. Actual: 1 hour
    - The user can select to filter dealerships by if they have cars available when browsing. Thus saving the time of physically visiting a dealership or calling in
3. Cathy hates when ads are targeted at her after visiting a site once. This has made Cathy avoid shopping for cars in the past. Cathy would like a place where she can view the data without the risk of being attacked with targeted ads.
    a. Estimate: 4 hours
    b. Actual: 5 hours
    - Completed by establishing a web-scraper to pull the data from sites that track users with targeted ads. The data is then used from a database instead of accessing the original website. This protects the user from targeted ads as The Drag is not ad supported and does not collect information on users.
4. Cynthia is a cybersecurity researcher. She has never purchased a car before and is looking for a new car to share with her partner Denver. Cynthia has heard about shopping sites showing different prices to different users based on their browser, device, or location or other factors. She and Denver have observed this price difference in their own search and would like to know the real list price for a certain car, not just the price an algorithm decides based on varying user device factors.

    a. Estimate: 3 hours
    b. Actual: 3 hours
    - The web scraper for The Drag is ignorant to all of the varying devices and locations the modern car buyer to be browsing for their new ride. Due to this fact, the site will only display one price for the car, not varying for multiple factors to try and trick the user into paying more than they should.
5. Ira loves Toyota! He has only ever driven cars from Toyota and does not plan to stop that any time soon. Ira is wondering what years Toyota manufactured cars to help in his search for the perfect Toyota.
    a. Estimate: 3 hours
    b. Actual: 2 hours
    - The web scraper was expanded in functionality to scrape the years that the each make has manufactured cars. This is displayed with the make to allow the user to determine the years a make was actively manufacturing cars.

<u>PHASE II</u>

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 an 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 NHTSA.
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: 2 hours
    - 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.

<u>PHASE I</u>

11. 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 amount of things.
    a. Estimate: 4 Hours
    b. Actual: 7 Hours
    - User story essentially describes 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.
12. 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.

13. 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.
14. 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).
15. 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 NHTSA. Involved making servlets and making helper functions and instantiating them for every JSP model file.

Customer User Stories:

PHASE III

1. As a car marketing junkie, I want to know if a car company with a logo I saw online is selling cars in my area.
    a. Estimate: 1 hour
    b. Actual: 0.5 hours

- This user story was resolved by adding a data source from our web-scraper to pull png images from carlogos.com and associate them with specific makes.

2. As a car shopping customer looking for a vendor listing a car of a specific name, I want to be able to search a specific car type and see where that car is being sold.
    a. Estimate: 2 hours
    b. Actual: 1 hours
    - Implementing this functionality was a result of cross-referencing our MongoDB database under dealerships, while displaying information about a specific car instance.
3. I found a car that I want to buy at a dealership. Since I just moved into the area, I don't really know my way around town. I want to be able to see the dealership's location on a map.
    a. Estimate: 2 hours
    b. Actual: 1 hours
    - This user story describes a desire to see an embedded map on each dealerships page. We solved this issue by using the google maps embed API.
4. As an avid car shopper, I want to be able to go back to square 1 on a car-shopping website after I have applied a lot of different filters and search queries.
    a. Estimate: 0.5 hours
    b. Actual: 0.5 hours
    - This just involved writing a button that would reload the search page with no passed forms, thereby resetting the page and clearing all filters/search queries.
5. As a first time car buyer, I don't want to waste time going to a dealer that isn't currently listing any cars. I want the ability to only display dealerships that are currently offering cars for sale.
    a. Estimate: 1 hour
    b. Actual: 1 hour
    - Satisfying this user story involved reading the car listing attributes of each of our dealership instances in MongoDB, and modifying the website display depending on if cars were listed or scraped from our API.

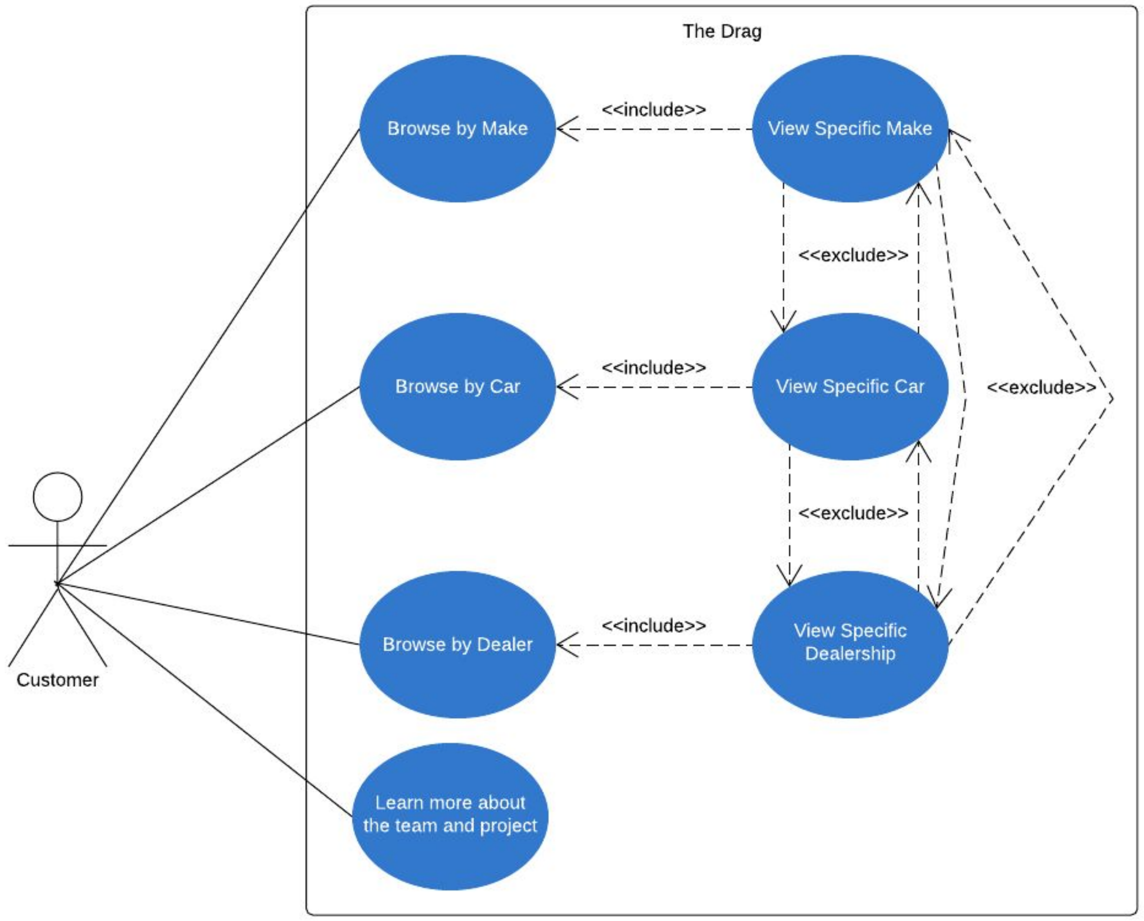
<u>PHASE II</u>

N/A

PHASE I

6. 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
7. 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.
8. 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.
9. 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.
10. I would like to know the closest locations of dealerships selling a certain make.
    a. Estimate: 3 Hours
    b. Actual: 1 hour
    - User describes the functionality the website can give to them.

Use Case Diagram



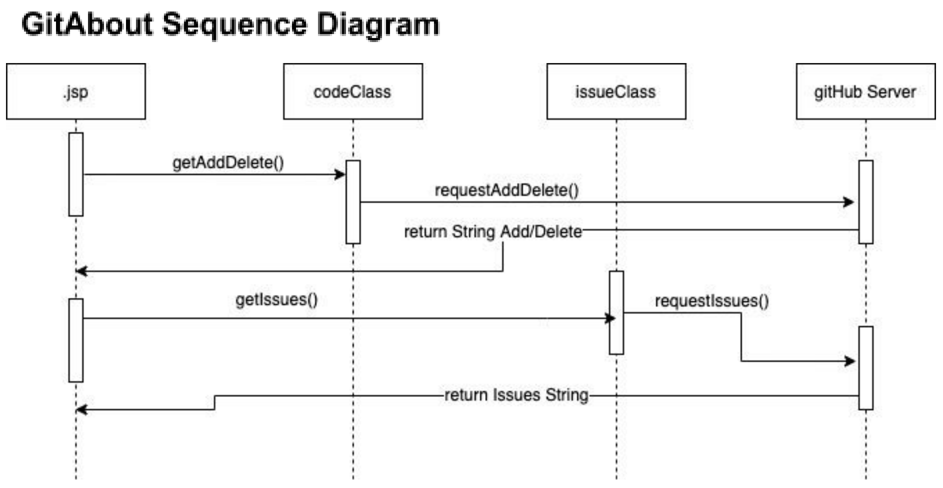
This UML Use Case diagram shows how users will interact with our site and browse through all of the three models. The User can initially select one of four options. Three of them will give the user a listing of instances to pick. After picking an instance they can learn more about it or use page links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

Design

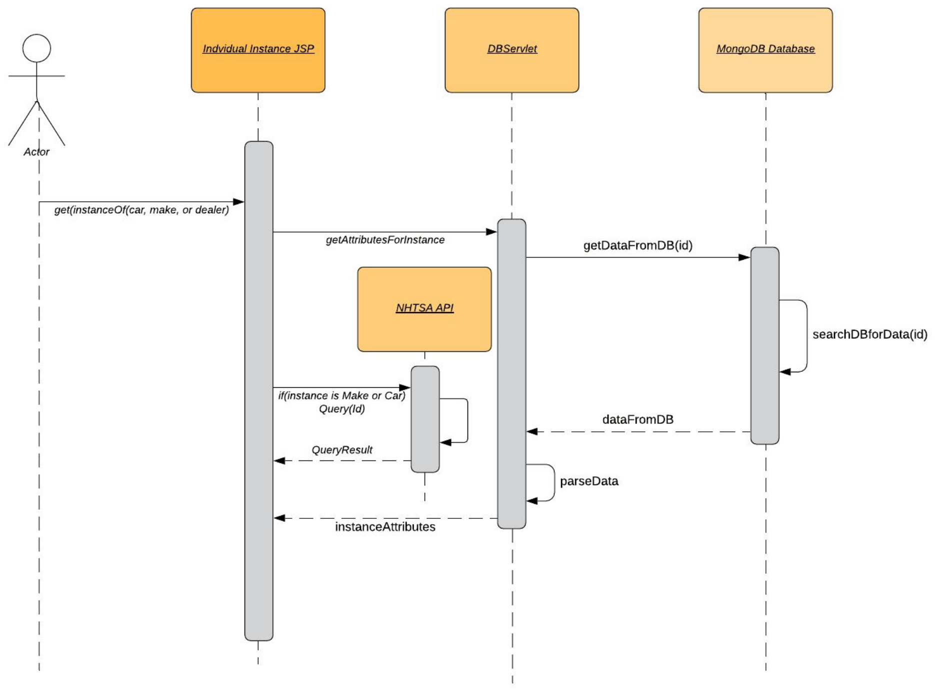
Our design was essentially giving users the options of choosing to browse by make, specific car, and dealership. Clicking “browse by make” will take you to a list of makes available. Clicking a specific make shows a list of car instances that are of that make to click. Clicking a specific car on that 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 can press "about" at the footer of the webpage.

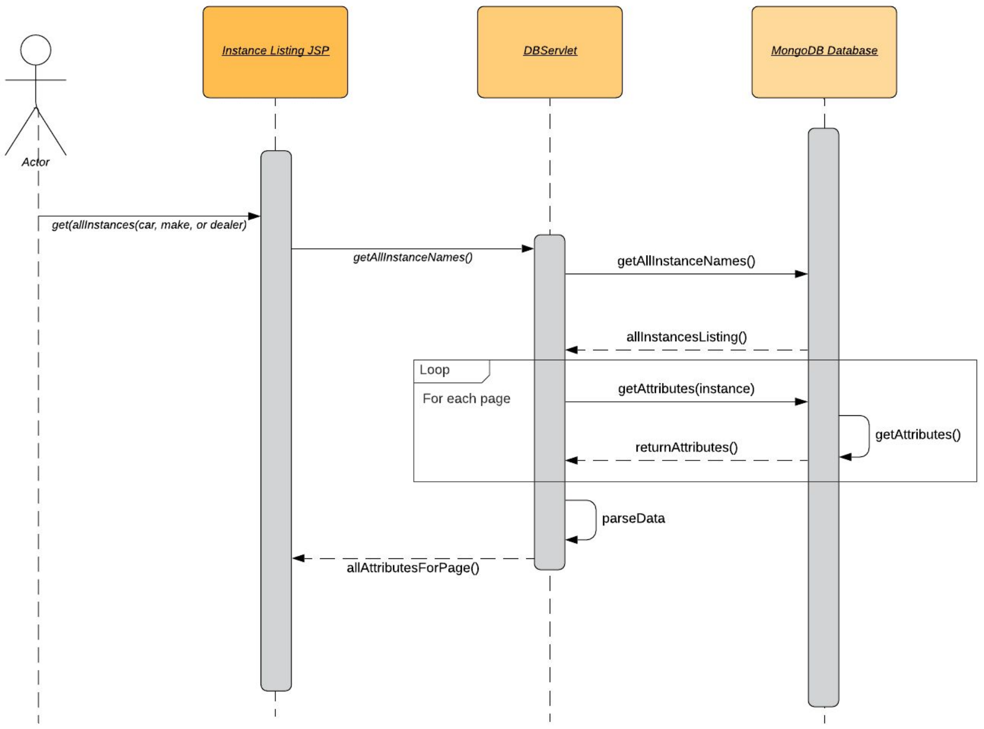
Responding to a comment from Phase I: The site logo functions as a home button, following the convention from most other websites. There is also a secondary "Home" link at the bottom of every page as a backup.



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.

Sorting, Searching, and Filtering

We have created multiple different ways to parse through instances on The Drag. Searching, sorting and filtering can be used in combination with one another for all model pages except for cars. We believe this combinatorial feature was not necessary for this phase because it didn't state it as a requirement in the TeamProject PDF. However, we were able to implement the combinational search, filter, and sort for makes and dealerships. Cars could not be implemented due to having over 11000 instances and a non-negligible runtime. Cars is different because of this fact and because we have to do a two level access in order to implement searching, sorting, and filtering for it. In order to search, sort, and filter cars we need to access the car's name attribute not its database id which is the VIN, so it is not as simple as doing an O(1) access. To reiterate, searching, sorting, and filtering are implemented for all models, but only makes and dealerships can use a combination of all three.

All searching for every model returns valid results based on their name i.e you can not search for a string that matches an attribute of a model and get those results. If you want to search for a specific attribute use the filter not the search bar (although not all attributes are filterable). The search bars are reserved for searching by name. This means if I searched "a" on makes it would return "Honda, Toyota, ***a, etc" not corresponding makes that have a market attribute that contain "a" in their string i.e "Luxury Cars, Sports Cars, etc". The dealers listing page allows the user to: sort A-Z & Z-A, filter dealerships to only ones with cars listed, filter dealerships by car makes they specialize in, and search dealerships. The cars listing page allows the user to: sort newest to oldest and vice versa, filter by manufacture year, and search the cars. The makes listing page allows the user to sort alphabetically from A-Z and Z-A, filter makes that are populated with cars on our website, filter makes by their market attribute, and search for a specific make.

Searching is done by converting the user inputted string to lowercase, and then converting our database documents ids for that model into lowercase as well. We then go through the database and see if any of the ids/names of the instances contain the search term and we return those results. Searching works a bit differently for car instances since they are keyed by their VIN. Instead we had to do a two level access to get their actual name attribute to compare with the search term.

Filtering is done by going through the database and returning an array of documents/instances that contain the requested attribute within their fields.

Sorting is done by calling collections.sort and collections.reverse on arrays that contain the database documents inside. In the case of cars, with over 11000 instances and because we had to do a two level access as mentioned above the runtime was extremely long. As a result we sort by oldest to newest car and newest to oldest car. We accomplished this by looking at each Car instance's VIN (unique ID). By looking at only the 10th character in each car's string we could decode and know what year it was manufactured in. We used this to sort the car instances. We also used this to filter cars by year.

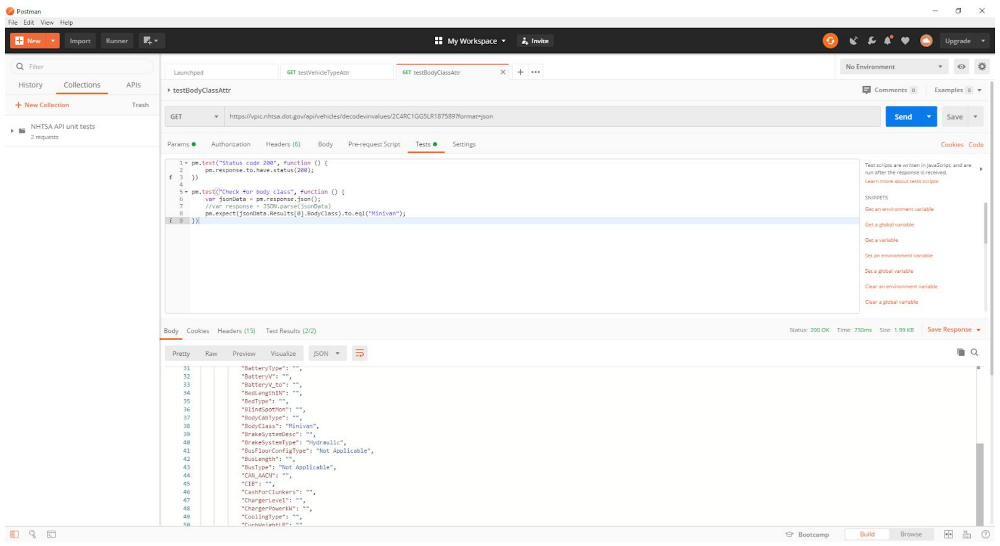
Testing

All unit test files can be found in src/test

<u>Backend: Total = ~ 116 Assertions and ~ 26 unit tests</u>

Postman used to test API calls:

testBodyClassAttr


GET https://vpic.nhtsa.dot.gov/api/vehicles/decodevinvalues/2C4RC1GG5LR187589?format=json

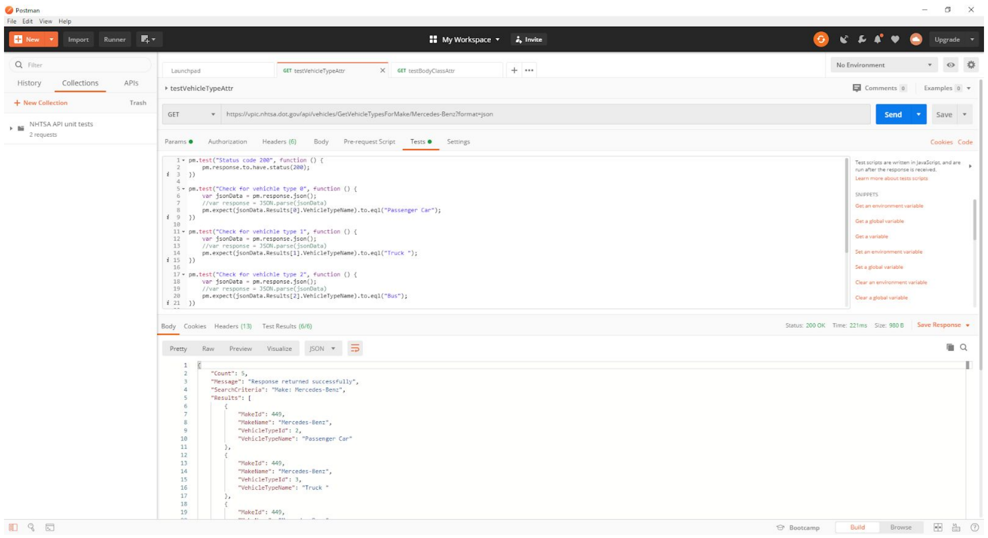
```
pm.test("Status code 200", function () {
    pm.response.to.have.status(200);
})

pm.test("Check for body class", function () {
    var jsonData = pm.response.json();
    //var response = JSON.parse(jsonData)
    pm.expect(jsonData.Results[0].BodyClass).to.eql("Minivan");
})
```

```
"BatteryType": "",
"BatteryV": "",
"BatteryV_to": "",
"BedLengthIN": "",
"BedType": "",
"BlindSpotMon": "",
"BodyCabType": "",
"BodyClass": "Minivan",
"BrakeSystemDesc": "",
"BrakeSystemType": "Hydraulic",
"BusFloorConfigType": "Not Applicable",
"BusLength": "",
"BusType": "Not Applicable",
"CAN_AACN": "",
"CIB": "",
"CashForClunkers": "",
"ChargerLevel": "",
"ChargerPowerKW": "",
"CoolingType": "",
```

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



GET https://vpic.nhtsa.dot.gov/api/vehicles/GetVehicleTypesForMake/Mercedes-Benz?format=json

```
pm.test("Status code 200", function () {
    pm.response.to.have.status(200);
})

pm.test("Check for vehichle type 0", function () {
    var jsonData = pm.response.json();
    //var response = JSON.parse(jsonData)
    pm.expect(jsonData.Results[0].VehicleTypeName).to.eql("Passenger Car");
})

pm.test("Check for vehichle type 1", function () {
    var jsonData = pm.response.json();
    //var response = JSON.parse(jsonData)
    pm.expect(jsonData.Results[1].VehicleTypeName).to.eql("Truck ");
})

pm.test("Check for vehichle type 2", function () {
    var jsonData = pm.response.json();
    //var response = JSON.parse(jsonData)
    pm.expect(jsonData.Results[2].VehicleTypeName).to.eql("Bus");
})
```

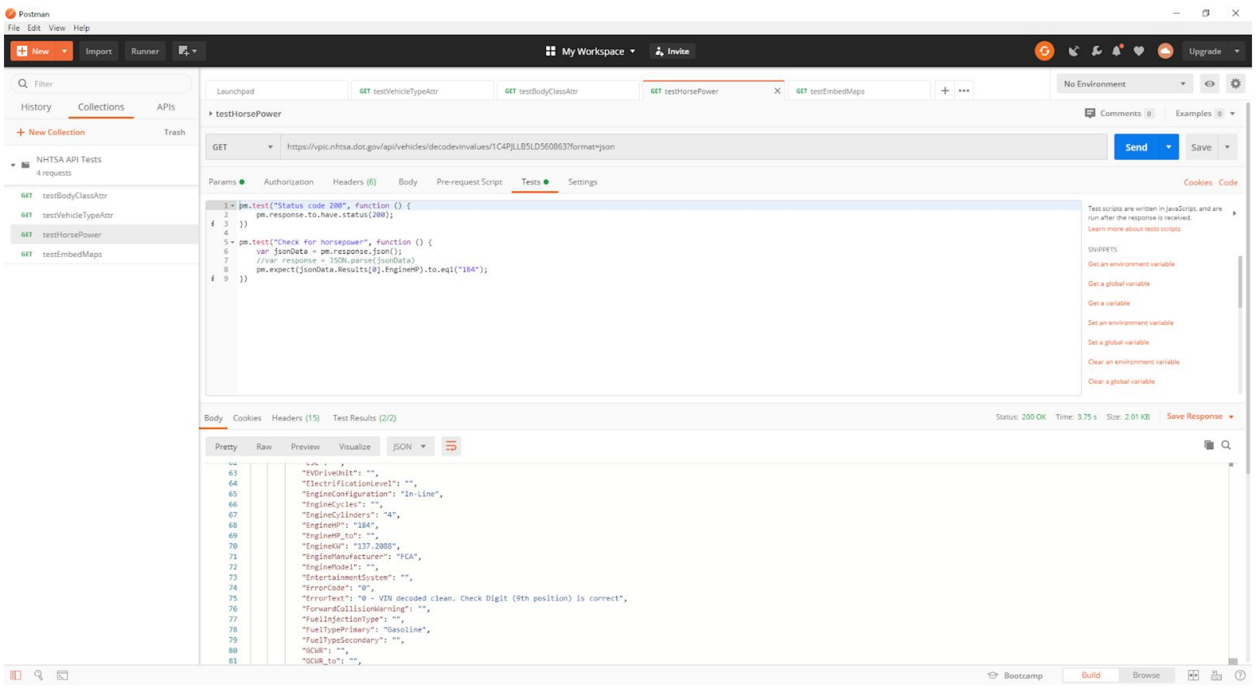
Status: 200 OK Time: 221ms Size: 980 B

Test Results (6/6)

```
{
    "Count": 5,
    "Message": "Response returned successfully",
    "SearchCriteria": "Make: Mercedes-Benz",
    "Results": [
        {
            "MakeId": 449,
            "MakeName": "Mercedes-Benz",
            "VehicleTypeId": 2,
            "VehicleTypeName": "Passenger Car"
        },
        {
            "MakeId": 449,
            "MakeName": "Mercedes-Benz",
            "VehicleTypeId": 3,
            "VehicleTypeName": "Truck "
        },
        {
            "MakeId": 449,
```

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.

testHorsePower



testHorsePower

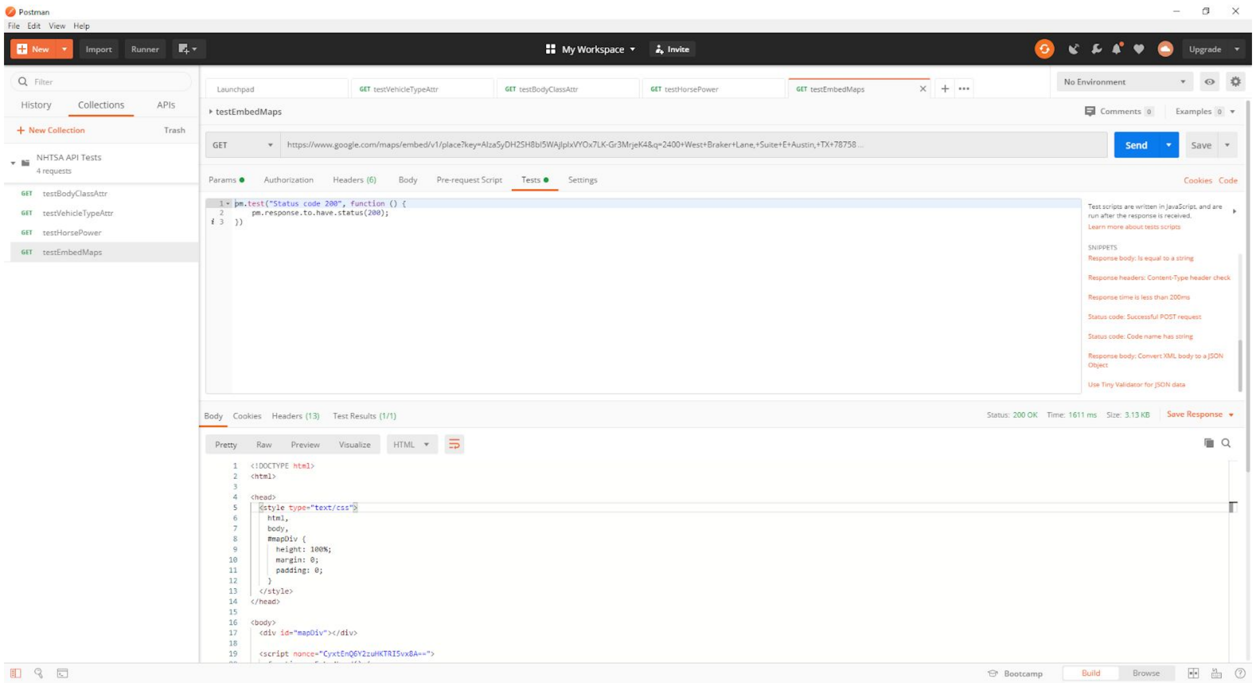
GET https://vpic.nhtsa.dot.gov/api/vehicles/decodevinvalues/1C4PJLLB5LD560863?format=json

```
pm.test("Status code 200", function () {
    pm.response.to.have.status(200);
})

pm.test("Check for horsepower", function () {
    var jsonData = pm.response.json();
    //var response = JSON.parse(jsonData)
    pm.expect(jsonData.Results[0].EngineHP).to.eql("184");
})
```

```
            "EVDriveUnit": "",
            "ElectrificationLevel": "",
            "EngineConfiguration": "In-Line",
            "EngineCycles": "",
            "EngineCylinders": "4",
            "EngineHP": "184",
            "EngineHP_to": "",
            "EngineKW": "137.2088",
            "EngineManufacturer": "FCA",
            "EngineModel": "",
            "EntertainmentSystem": "",
            "ErrorCode": "0",
            "ErrorText": "0 - VIN decoded clean. Check Digit (9th position) is correct",
            "ForwardCollisionWarning": "",
            "FuelInjectionType": "",
            "FuelTypePrimary": "Gasoline",
            "FuelTypeSecondary": "",
            "GCWR": "",
            "GCWR_to": "",
```

Description:
This test is used to check if we can successfully call the NHTSA API and to check if we can get theEngineHP of horsepower 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 horsepower.

testEmbedMaps


testEmbedMaps

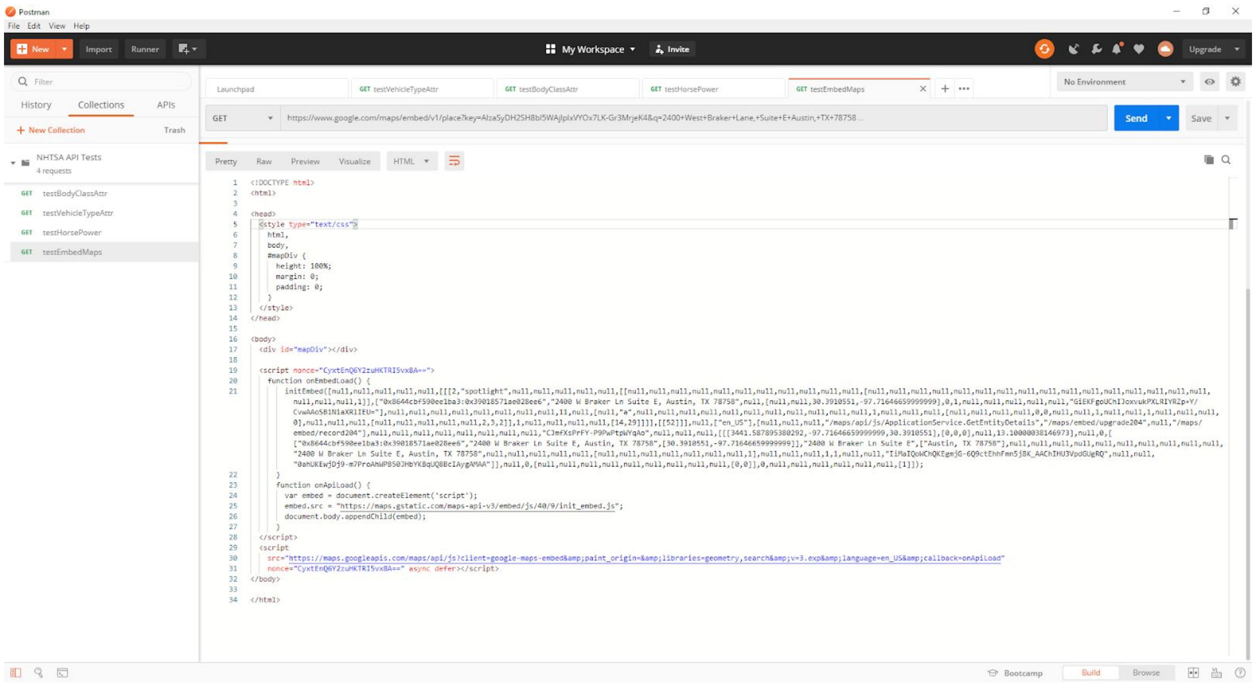
GET https://www.google.com/maps/embed/v1/place?key=AIzaSyDH2SH8bI5WAjlplxVYOx7LK-Gr3MrjeK4&q=2400+West+Braker+Lane,+Suite+E+Austin,+TX+78758 ...

Tests:

```
pm.test("Status code 200", function () {
    pm.response.to.have.status(200);
})
```

Body — Test Results (1/1) — Status: 200 OK Time: 1611 ms Size: 3.13 KB

```
<!DOCTYPE html>
<html>

<head>
  <style type="text/css">
    html,
    body,
    #mapDiv {
      height: 100%;
      margin: 0;
      padding: 0;
    }
  </style>
</head>

<body>
  <div id="mapDiv"></div>

  <script nonce="CyxtEnQ6Y2zuHKTRI5vx8A==">
```


GET https://www.google.com/maps/embed/v1/place?key=AIzaSyDH2SH8bI5WAjlplxVYOx7LK-Gr3MrjeK4&q=2400+West+Braker+Lane,+Suite+E+Austin,+TX+78758 ...

```
<!DOCTYPE html>
<html>

<head>
  <style type="text/css">
    html,
    body,
    #mapDiv {
      height: 100%;
      margin: 0;
      padding: 0;
    }
  </style>
</head>

<body>
  <div id="mapDiv"></div>

  <script nonce="CyxtEnQ6Y2zuHKTRI5vx8A==">
    function onEmbedLoad() {
      initEmbed([null,null,null,null,null,[[[2,"spotlight",null,null,null,null,null,[[null,null,null,null,null,null,null,null,null,null,null,[null,null,null,null,null,null,null,null,null,null,null,null,null,null,null,
        null,null,null,1]],["0x8644cbf590ee1ba3:0x39018571ae028ee6","2400 W Braker Ln Suite E, Austin, TX 78758",null,[null,null,30.3910551,-97.71646659999999],0,1,null,null,null,null,null,"GiEKFgoUChIJoxvukPXLRIYRZp+Y/
        CvwAAoSB1N1aXRlIEU="],null,null,null,null,null,null,null,null,11,null,[null,"a",null,null,null,null,null,null,null,null,null,null,null,1,null,null,null,[null,null,null,null,0,0,null,null,1,null,null,1,null,null,null,
        0],null,null,null,[null,null,null,null,null,2,3,2]],1,null,null,null,null,[14,29]]]],[[52]]],null,["en_US"],[null,null,null,"/maps/api/js/ApplicationService.GetEntityDetails","/maps/embed/upgrade204",null,"/maps/
        embed/record204"],null,null,null,null,null,null,null,null,"CJmfXsPrFY-P9PwPtpWYqAo",null,null,null,[[[3441.587895380292,-97.71646659999999,30.3910551],[0,0,0],null,13.10000038146973],null,0,[
        ["0x8644cbf590ee1ba3:0x39018571ae028ee6","2400 W Braker Ln Suite E, Austin, TX 78758",[30.3910551,-97.71646659999999]],"2400 W Braker Ln Suite E",["Austin, TX 78758"],null,null,null,null,null,null,null,null,null,null,
        "2400 W Braker Ln Suite E, Austin, TX 78758",null,null,null,null,null,[null,null,null,null,null,null,null,1],null,null,null,1,1,null,null,"IiMaIQoWChQKEgmjG-6Q9ctEhhFmn5j8K_AAChIHU3VpdGUgRQ",null,null,
        "0ahUKEwjDj9-m7ProAhWPB50JHbYKBqUQ8BcIAygAMAA"]],null,0,[null,null,null,null,null,null,null,null,null,[0,0]],0,null,null,null,null,null,null,[1]]);
    }
    function onApiLoad() {
      var embed = document.createElement('script');
      embed.src = "https://maps.gstatic.com/maps-api-v3/embed/js/40/9/init_embed.js";
      document.body.appendChild(embed);
    }
  </script>
  <script
    src="https://maps.googleapis.com/maps/api/js?client=google-maps-embed&amp;paint_origin=&amp;libraries=geometry,search&amp;v=3.exp&amp;language=en_US&amp;callback=onApiLoad"
    nonce="CyxtEnQ6Y2zuHKTRI5vx8A==" async defer></script>
</body>

</html>
```

Description:
This test is used to check if we can successfully call the Google Embed Maps API and we check the outputted JSON to ensure the address and API key were passed successfully. Each dealership instance has a unique address which we can use to make the call to the api for that specific dealership's location on the map.

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:
We refined the front-end testing from the last phase to now include the new GUI elements added with the sorting, filtering, and searching functionalities. The actual testing implementation is the same as the last phase in which we utilized Selenium IDE testing. Once again downloading the developer version of Selenium as a Google Chrome extension and “recording” clicks/user GUI inputs. In this case now including user text input from a keyboard. We then translated them to JUnit java files with slight modifications such as adding waits for 15 seconds to allow all elements of a page to load properly and give the database appropriate time to fetch data. The main focus with each of the tests was to check if the new sorting, filtering, and searching GUI bars worked properly and you could still access the different and expected pages. It was also a focus because the cars model page had a different set up for the sorting, filtering, and searching GUI bar than the makes and dealerships pages. Selenium helped test the searching function/HTML form of each of the model pages and checking if our own

parsers we set up to deal with user inputs worked properly. One issue that came across that would fail test cases that could not be avoided was when the NHTSA API was down, which is also covered in this report, but was only temporary.

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/
Source 4: Google Maps Embed API

Note about sources:
Other than the Google Maps Embed API, the other APIs/Sources are subject to the whims of their website owners. Cars.com is a live car selling website so our database might not be the same as Cars.com after some time because we scrape data from the website instead of making an API call to the site. We do this because cars.com does not have a free API which to call. How this affects our website is by affecting our car instance image links and urls links since they are tied as an href to cars.com. If a car is sold it will be taken down and thus the purchase link and image link will go down. Despite this, we are sure that it will only affect a minor portion of the over 11,000 cars we have scraped. In addition, there is also a possibility that carlogos.org will change, which will also affect our make urls and images. The chance of this is uncertain, it is only a possibility because during the time between Phase 2 and 3 the website changed its DOM layout forcing us to fix the web scraper. In our many tests of API calls the NHTSA API failed to connect only on 4/22/2020 sporadically throughout the day, on

other days it worked without issue. This is caused because the actual API is down for maintenance. When this happens, the user will be unable to look at a make instance (Honda) or a car instance (2020 Honda CIvic...) because they make calls to NHTSA. A NULLPointerException will occur because they are unable to connect the API. We humbly request in the case that this happens for you to wait for an hour and try again. In the end, this issue is caused by the API itself going down and not because our website is not fully functional.

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 11000 cars on our site.

Makes are generated by bucketing the data scraped from carlogos.org by make. Images for each make are gathered from programmatically carlogos.org. Dealerships are also created via bucketing the individual cars. We also use the Google Maps Embed API to display an embedded map showing the location of a dealership.

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.

Note since our database criteria is only new cars within 20 miles of Austin some attributes will not be filled because of limitations. Some dealerships don't have a website, phone number, picture, and/or description because they don't list these attributes on the website we scrape from. In fact, our web scraper functions properly, it's just because these dealerships don't post these properties on cars.com. In addition, some dealerships only sell used cars and thus have no new phone number, only a used phone number, vice versa. Some makes are not sold within 20 miles of Austin so they are not sold at any dealers and have no cars. Even so our criteria gives us over ~11,000 instances of cars, ~170 dealerships, and ~300 makes

Attributes in red are the three attributes shown on the model listing pages. They are also shown on the specific instance's page.

Make Instance Attributes/Data -
Number of cars sold within 20 miles, number of dealerships within 20 miles, vehicle types, target market, years active, brand information link, cars available (link to other model), dealerships the make is sold at (link to other model)

Car Instance Attributes/Data -
Listing purchase website link, price, mpg, model year (implied), horsepower, body class, vin, name of seller, dealership (link to other model), make of car (link to other model)

Dealership Instance Attributes/Data -
Address, phone number for new cars (when applicable), phone number for used cars (when applicable), dealership website (when applicable), hours of operation, about description (when applicable), embedded google map, cars sold at dealership (link to other model), makes sold at dealership(link to other model)

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 has had good communication going throughout each phase. 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 practiced pair programming heavily 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 page 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.

In Phase III we truly had our group work style down and we got through the phase quickly. We continued to practice pair programming via Zoom and screen sharing, sometimes with our entire group watching and helping out. (Surprisingly, five programmers at once is significantly better than one.) We re-scraped our sources and updated the database to its final iteration. We also added sorting, filtering, and searching functionality to our model listing pages. Our homepage was also redesigned to be more user-friendly as well as more aesthetically pleasing. Overall we are very pleased with our site and we cannot wait to present it to the class.