FunFriends33

CS 327E

Milestone 7

**Ranger Motors Database Report**

**Introduction:** Our group, FunFriends33, saw the need for a program that would allow its user to discover summary information about the inventory of a local used car dealership, Ranger Motors Austin. To fill this need, we created a database model and used Python to scrape the dealership’s website to acquire information on the inventory, clean and organize the inventory data, store the data in a MySQL database, and create an interface allowing a user to query the database. Here, we describe the background and purpose of our project and our solution, then detail the process of creating each of the above components of the program. We then conclude by evaluating the final version of our project and reflecting on our development process.

**Background/Motivation:** Our project idea arose from the following observation: while Ranger Motors’ website allows inventory filtering that is helpful to a customer hoping to buy a car, it lacks some features that would be desired by a person hoping to learn about the nature of the inventory as a whole. Our hypothetical user, then, is someone who is interested in describing the inventory as a whole in terms of aggregations like summary statistics. This might be an employee of Ranger Motors who wants to employ marketing analytics to see what car features are most profitable, or perhaps a competitor hoping to discover the differences between Ranger Motors’ inventory and their own.[[1]](#footnote-1) Whereas the Ranger Motors website allows users to narrow down the results until they find “the perfect car,” we want to allow our user to run statistics on customized groups of cars. This could include finding the average price of all white Fords, or the number of cars from 2007 that have less than 100,000 miles on them. Such statistics cannot be found using the website’s inventory search. This, then, was our motivation for creating a data pipeline to store the inventory data and an interface to allow users to run custom queries on the database.

**Solution/Process:** Having defined our problem, we began to outline our solution. Naturally, we began by fleshing out the functionality described in the previous section. In addition to filtering results, we wanted our user to be able to sort results by attributes of the cars, choose the attributes to display in the results and, most importantly, group cars on certain attributes and display basic statistics (e.g., count, max, avg) for each of the groups. Further, we sought to extend the customizability of the attribute filtering, allowing user-entered values to be used to filter both the numeric and text-based attribute values.

We planned for each of the components of our project: we would start by using BeautifulSoup to scrape Ranger Motors’ website for inventory data. Next, we would use Python to clean and organize the data and store it in a MySQL database with PyMySQL. Finally, we would create a text-based user interface in Python to allow custom queries to be run on the database. In an effort to minimize the knowledge required by the user, we made sure that no knowledge of SQL syntax was required to make use of the program, although knowledge of relational database concepts are to some extent necessary in order to understand how to interact with the program.

**Database Design:** Figure 1 shows our initial conceptual diagram for the Ranger Motors Database. Originally our ERD consisted of three tables. The main table in this database is the CarsForSale table, which contains the bulk of the information on the inventory. Each record in this table corresponds to a vehicle in the Ranger Motors inventory, with attributes that describe the car. The primary key here is the VIN (vehicle identification number), as every vehicle made has a unique VIN.

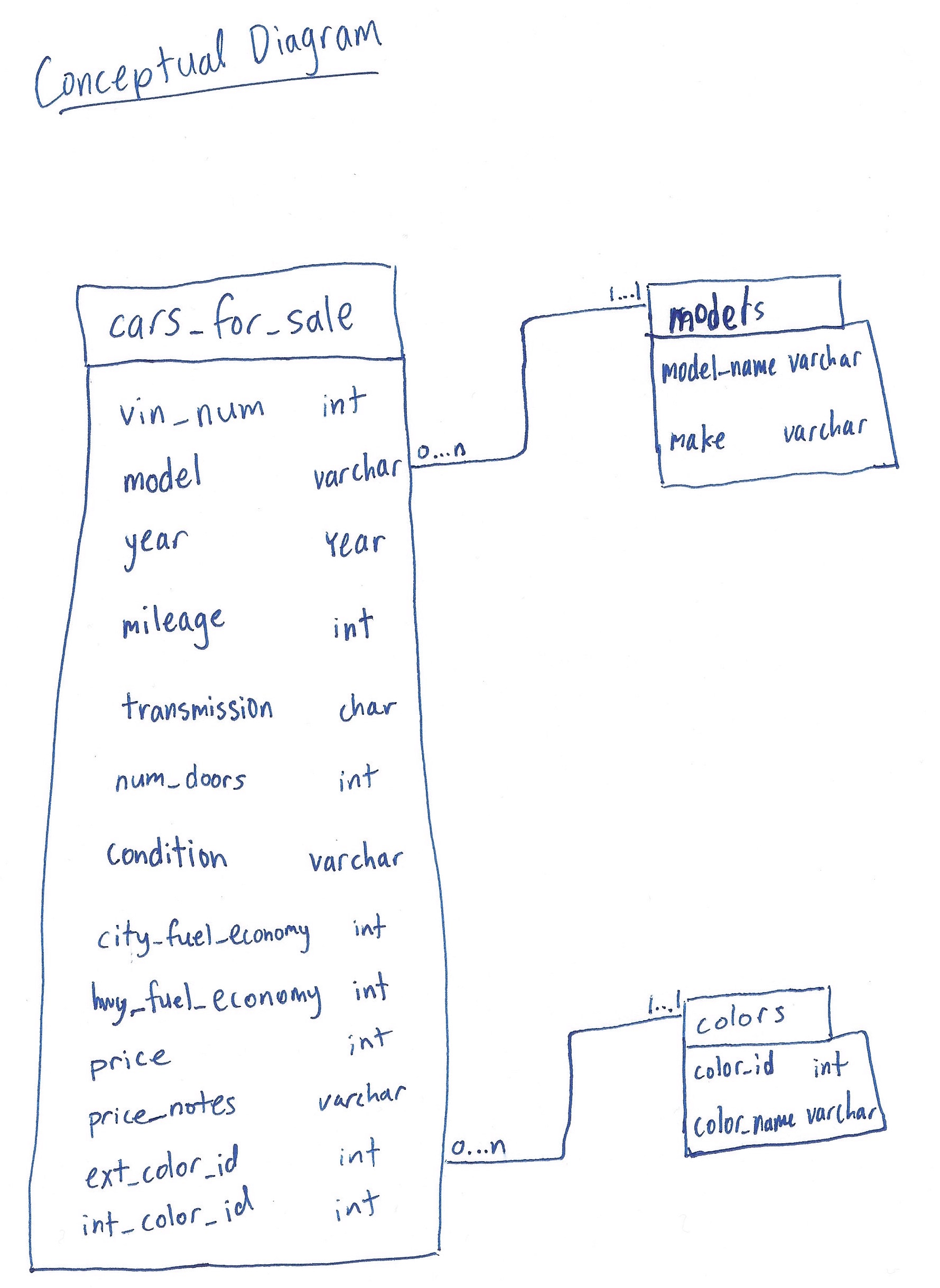


Figure 1: Original Conceptual Diagram

Our original diagram had two additional tables, the first being the *Makes/Models* table, to which the CarsForSale attribute *model* is a foreign key. We added this table to ensure that our CarsForSale table was in third normal form. Since we took *make* to be functionally dependent on *model*, we had to create another table to achieve this, hence the existence of the *Make/Models* table. Having our tables in third normal form was meant to reduce redundancy in our database.

The last table in our original diagram is the *Colors* table, to which the *CarsForSale* attributes *ext\_color\_id* and *int\_color\_id* are foreign keys. This table consists of color identification numbers, which are referenced by the external and internal color attributes for the vehicles, each of which matches the name of a color, in string form. We added this table for the purpose of validation: given that many different colors exist, using color IDs as the *CarsForSale* attributes, rather than the color names themselves would ensure that all values input matched to legitimate colors. This would prevent erroneous input, so that we could not accidentally end up with “multiple versions” of a given color, (e.g., “red,” “Red,” and “redd”).

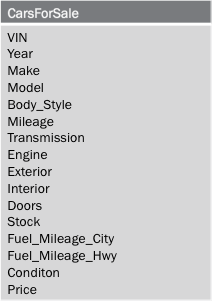


Figure 2: Revised Conceptual Diagram

Figure 2 depicts our revised conceptual diagram, which consists of a single table. While the main table from the original diagram remains in the revised edition, the *Makes/Models* and *Colors* tables were removed. We decided to remove the *Makes/Models* table because having the table really did not add any value to our database. The avoidance of redundancy may have been useful had we been dealing with a much larger database, but redundancy in our database is not an issue; the Ranger Motors inventory consists of fewer than 100 vehicles, which means that the *CarsForSale* table has less than 100 records. We concluded that the addition of the *Makes/Models* table added complexity to no clear benefit.

We also removed the *Colors* table in the revision of our model, as we found validation to be unnecessary. Validation of input for the color attributes would be useful if our data was being manually entered. However, given that we are storing our data automatically using a web-scraping program, the use of a validation table would have no purpose. When checking values against the validation table, our only option for a value not in the table would be to either add it to the table or leave the value as Null. Either way, using such a table would not benefit our database. So we decided to leave the *Colors* table out of our model.

These revision decisions leave us with a model consisting of a single table. This raises the question of whether using a relational database makes any sense for this project. Aside from the obvious factor (that this project is for a Databases course), we think that using a relational database makes sense here. While the information, as is, does not require a relational model, we can only benefit from having the flexibility to add relational components in the future. For example, the use of a relational database, as opposed to spreadsheet software, leaves us the possibility of adding inventory information for other Ranger Motors Dealership locations. Moreover, we retain all of the necessary aggregation and statistical functionality in using a relational database, so using a spreadsheet or other software would not give us any additional benefit in that sense. Thus, we believe that our use of a relational database is legitimate.

**Web Scraping:** We scraped the Ranger Motors Austin website to get all of our information on the vehicle inventory. The website has an inventory page that is structured such that it contains the listings for each of the vehicles with some summary information and a picture for each vehicle. The inventory page is shown in Figure 3. Each of these can be clicked (i.e., are hyperlinked) to the individual webpages for the respective vehicles. The individual vehicle pages have more pictures and more complete information on the vehicle. An example of one of these pages is shown in Figure 4.

We used Python’s URL Library with the URL of the Inventory page to get the html and then used BeautifulSoup to parse this html and find the URLs of all of the individual vehicles’ webpages. This was simple once we discovered that a class, “inv-view-details,” existed for such links: we used BeautifulSoup to find all hyperlinks with this class tag. Once on a particular vehicle’s webpage, we had to retrieve information from a few different places. As is clear from Figure 4, most of the attribute information is contained in a table. We got the Attribute names, the left side of this table, by finding matching items with the “list-group” class with our BeautifulSoup object. Then we got the particular values for these attributes by recognizing that the values had the class “list-group-item.” We created Python dictionaries for each vehicle listing in order to store each of these key-value pairs. Using dictionaries resolved the issue that not all listing pages have tables that match exactly on their Attributes information; dictionaries allowed us to store the Attribute names and values that appeared on each page, rather than having to hard-code a pre-defined set of Attribute names.

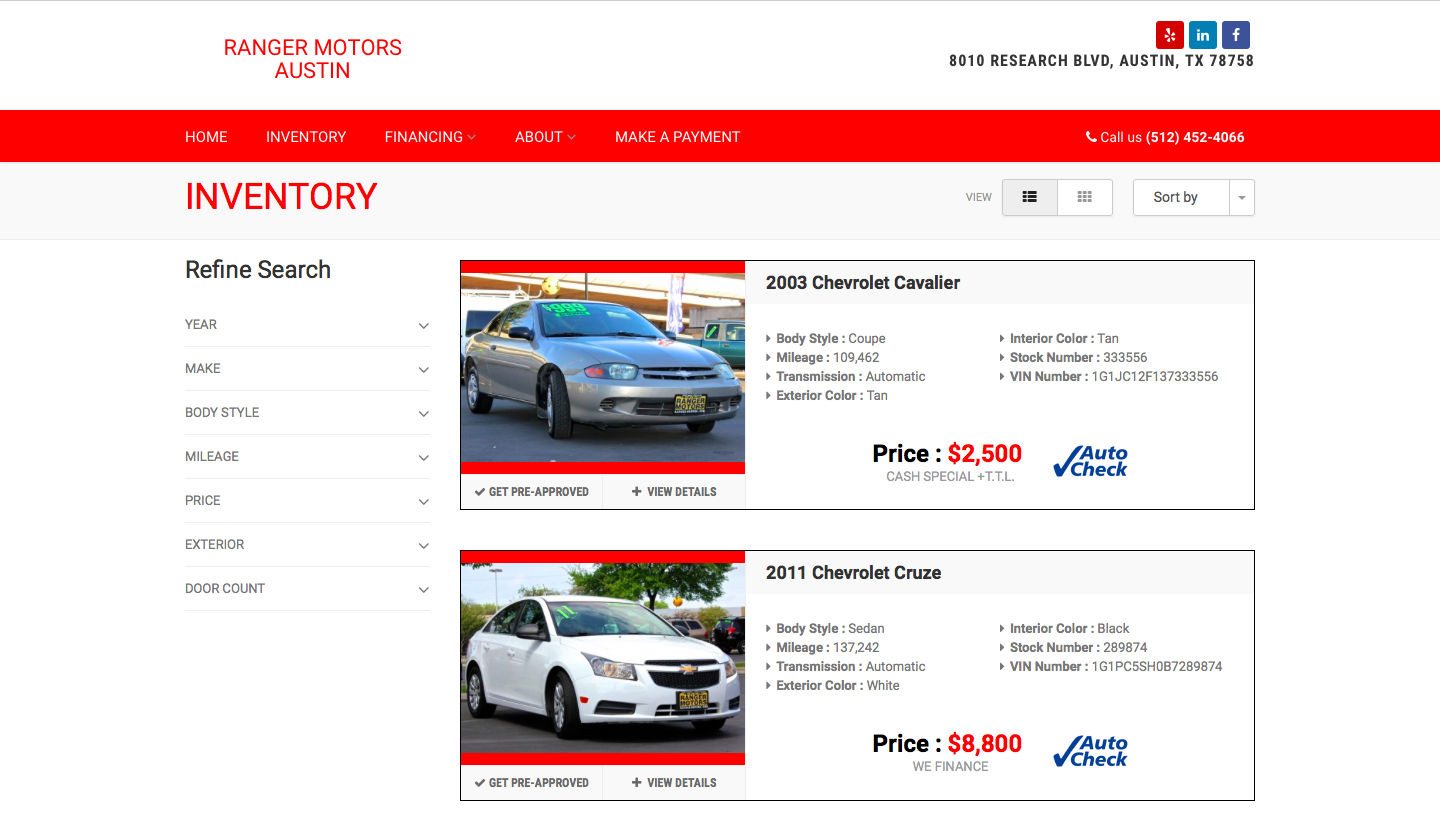


Figure 3: Ranger Motors Austin Inventory Webpage (http://www.rangermotorsaustin.com/inventory/)

In addition, we had to get the price of the car from the bubble at the top right. This required us to search for a separate class, “details-price,” which we then parsed in Python to get the integer value for the price.

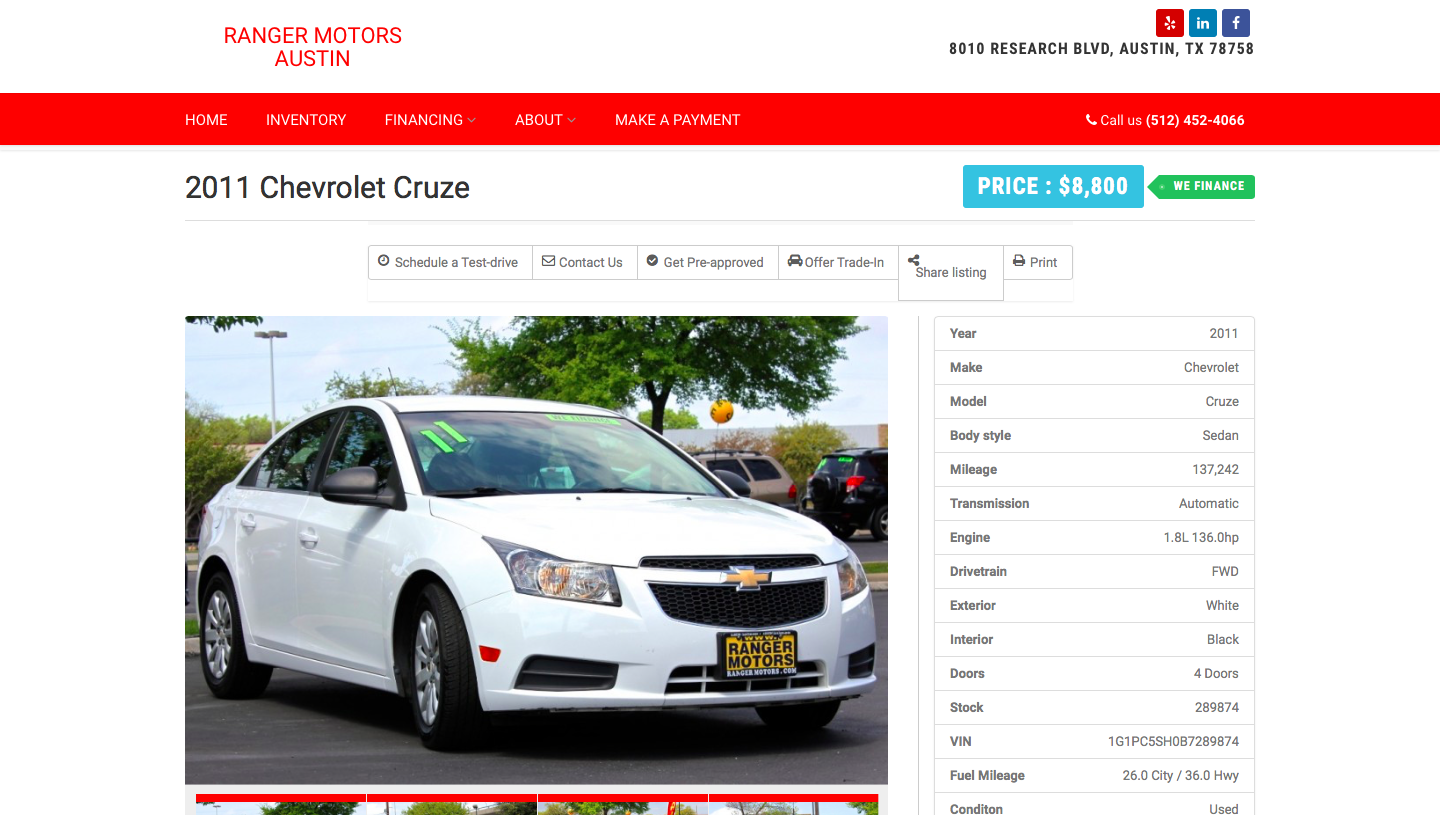


Figure 4: Sample Webpage for Individual Vehicle

1. This shit is whack, my dawg. [↑](#footnote-ref-1)