

**Final Project: Car Dealership**

**CS157A: Introduction to Database Management Systems**

Dr. Jahan Ghofraniha

By: Yusuf Mostafa, Leo Alciso, Patrick Merrill

## Table of Contents

Executive Summary.....	3
Background/Introduction.....	3
Problem Statement.....	3
Purpose/Motivation.....	4
Design.....	4
Implementation and Testing.....	12
Conclusion.....	17
Appendix.....	18

## **Executive Summary**

Vehicle inventory management presents a problem for database engineers. While there are commonalities in vehicle archetypes, there are also anomalies and brand discrepancies that pose issues when compacting a large group of vehicles into a comprehensive dataset. Through logical design and normalization, the data was condensed to a point of simple queries. Through conceptual design, a front that accessed this database was created to best serve the end user.

## **Background/Introduction**

Vehicle resale websites represent a number of interesting problems to explore in terms of relational database management. The large number of varying features for vehicles of the same make and model produce repeating data and independent attributes that need to be resolved through normalization. Vehicle resale also requires the tracking of vehicle sellers and transactions adding to the complexity of the design and normalization process. Additionally, facilitating the sale of used vehicles requires the ability to dynamically add data to the database, update or delete existing values, and join data from multiple tables to display pertinent information to users. These factors offer a unique challenge in the database design and implementation.

## **Problem Statement**

Inventory management is difficult for car dealerships due to the specific qualities in each vehicle. There are countless possibilities in options for a car, which calls for a normalized database design to help simplify a large inventory.

## Purpose/Motivation

CarGurus, AutoTrader, and Carmax are a few popular sites that allow someone to search for cars to purchase. They all vary in their approach in searching and querying through hundreds of thousands of listings. This project starts from the ground-up to take vehicles, along with their many specific features, and put them into a database which was then normalized for simpler queries and lookup.

## Design

### Conceptual Design

The conceptual design began with the landing page that proceeds a successful login. This page would show all available listings of vehicles for sale. From here, the user will be able to either make a search query, view listing details, or view past transactions. Search results will customize the listings shown on the page. From there, the user may again view more detailed information on specific listings.

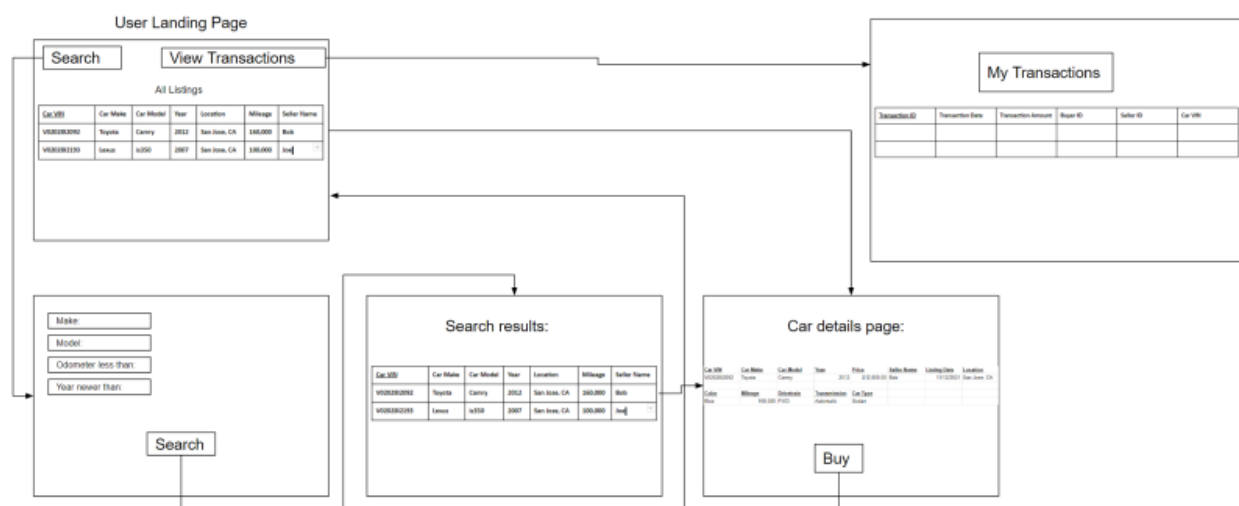
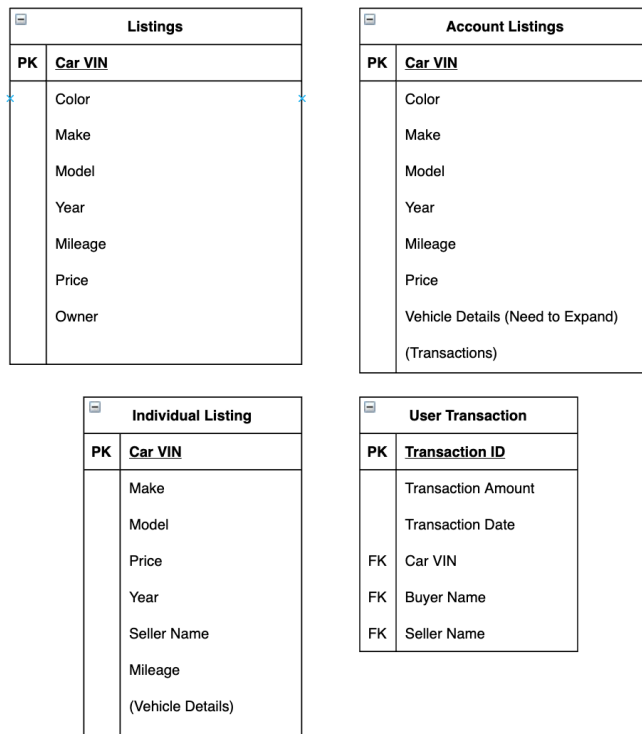


Figure 1 - Storyboard

The user views that will be seen by the end user are shown in the following ERD.



**Figure 2 - User Views ERD**

The User Stories capture features of a software from the perspective of a potential user. The first user story shows the flow from a new user perspective. The user will click the Register button that routes to a form. Within the form, they will enter their information and send the form by clicking Create Account. From there, they will reach the landing page with all available listings.

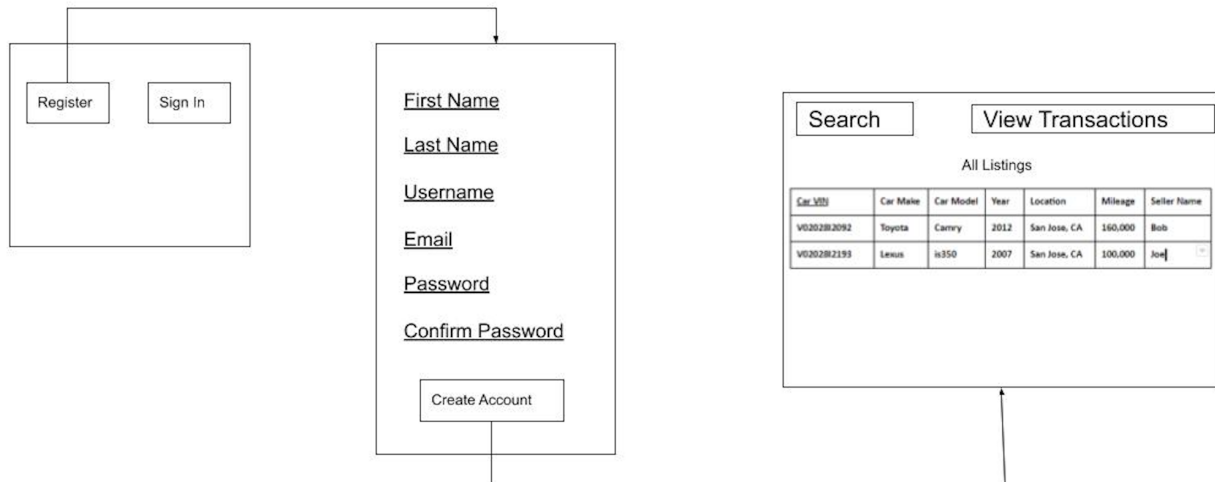
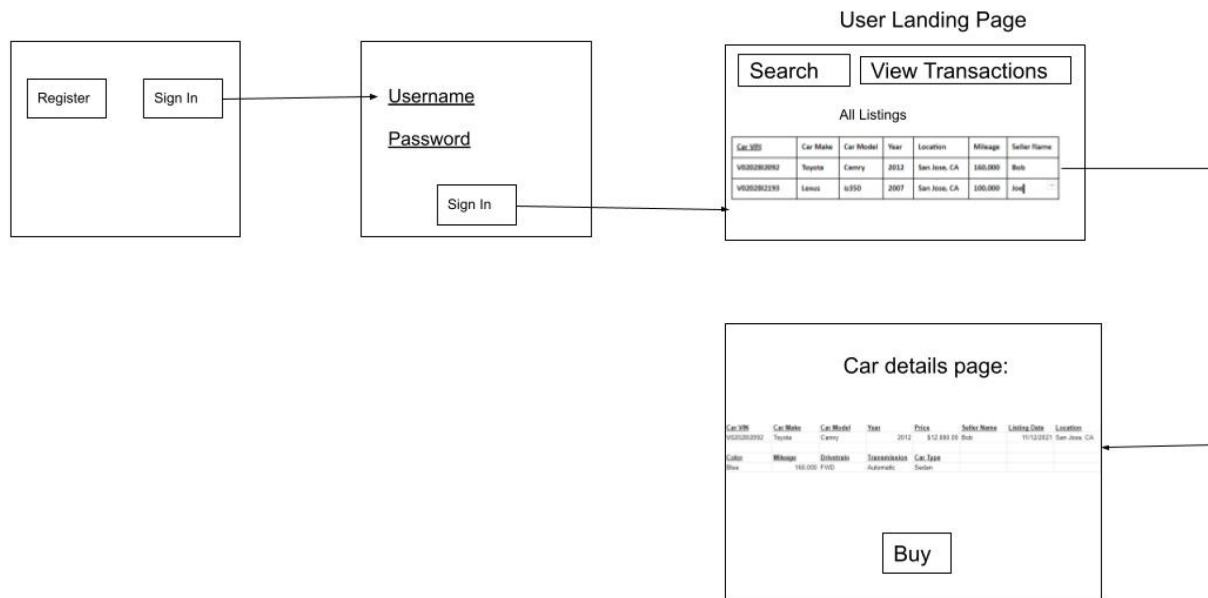


Figure 3 - User Story: New User

The next user story shows the steps to buy a car from a registered user perspective. The user chooses Sign In from the homepage, which routes to a form where they will input their credentials. After signing in, they will enter the landing page where they can see all available listings. From this page, the user will choose a listing to route to a page showing all of the specifics on the vehicle. If the user wants to purchase the vehicle, then they can click the Buy button.



**Figure 4 - User Story: Buying a Car**

Another user story is for a user that would like to view their past transactions. This is accomplished by choosing Sign In from the homepage, and entering their credentials in the Sign In form. After signing in, they are directed to the landing page. The landing page has a View Transactions button that can be clicked to route to a page that contains a table showing the past transactions of the logged in user.

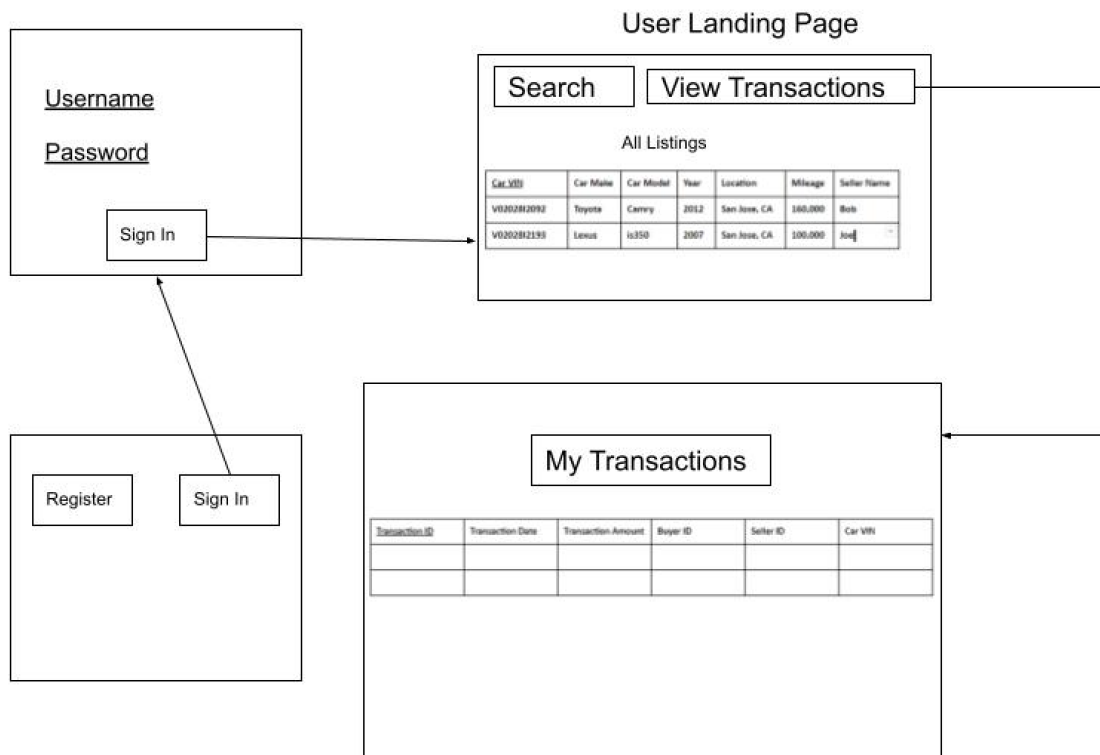


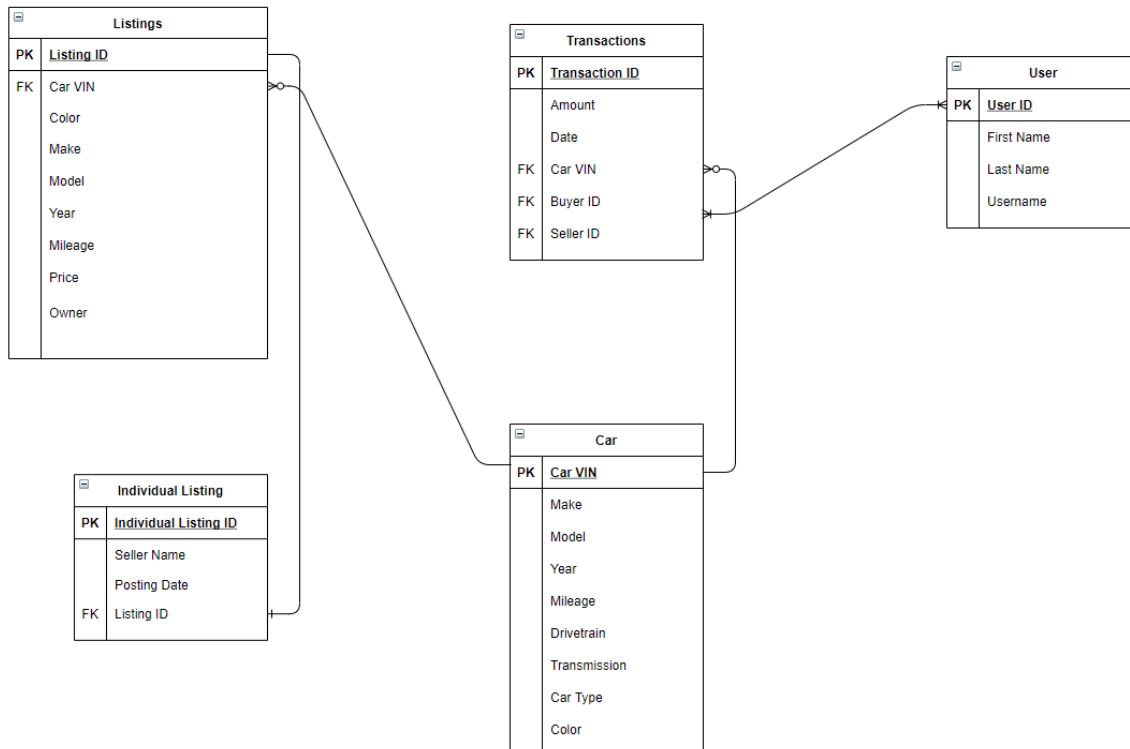
Figure 5 - User Story: View Transactions

## Logical Design

The logical design of the database was done through the normalization of the user views from the conceptual design. To remove repeating groups and multivalued attributes for first normalization, separate tables were made for the listing page, individual listings, transactions, cars, and users. The Listings table has a one-to-one relationship with the Individual Listing table as each row in the Listings will correlate to a row in Individual Listings containing more details about the seller and posting date of the listing. The Listings table is also related to the Car table through a one-to-many relationship as multiple listings may contain the same vehicle in the event that a vehicle is re-sold on the site. The Car table is related to the Transactions table through a one-to-many relationship as, similar to listings, multiple transactions may involve the same car if



a resale occurs. The Transactions table has a many-to-many relationship with the user table as each transaction will contain two users, the buyer and the seller, and each user may be part of multiple transactions.



**Figure 6 - First Normalized Form**

Removing partial dependencies was done to normalize the tables into second normal form. The attributes of color, make, model, year, and mileage were removed from the listings table as those attributes are dependent on the vehicle. The attribute of amount was also removed from the transaction table as that is functionally the same as price which is dependent on the associated listing. The seller ID was removed from the transaction table with the reasoning that this is fundamentally the same as the owner attribute in the listing table and was dependent on the listing. Additionally, the make attribute was removed from the car table since the make is

dependent on the model of the vehicle. Due to this, another table called Model was made with the model being the primary key and the make being an attribute. The model table has a one-to-many relationship with the car table as many cars can have the same make and model. The vehicle's VIN was also removed from the Transactions table as that attribute is contained in, and dependent on, the listing. The Listings table has a one-to-one optional relationship with the Transactions table since there will not be a transaction relating to a listing if a purchase has not taken place.

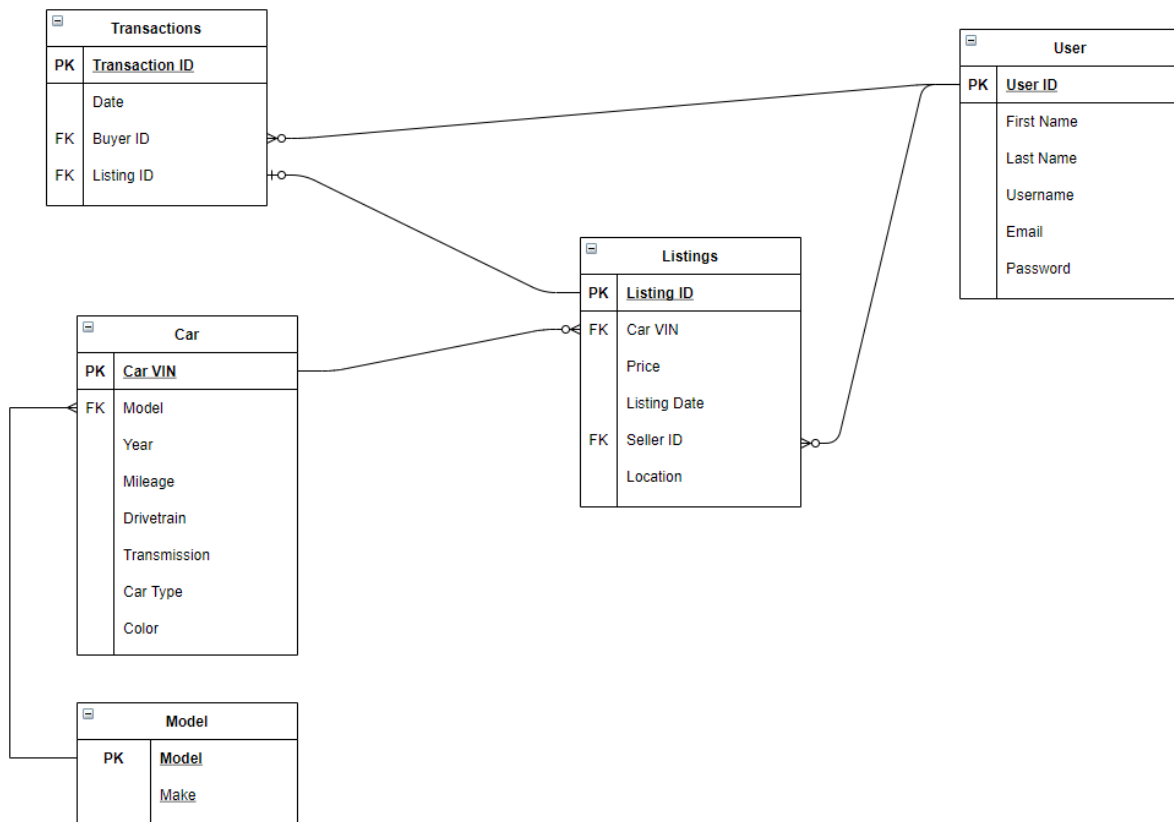


Figure 7 - Second Normal Form

To move into the third normal form, transitive dependencies were removed from the tables above. The drivetrain, transmission, and car type attributes were removed from the car

table as these are transitively dependent on the car vin through the model and year of the vehicle. This resulted in a new Model table with the Model and Year acting as a composite key for the drivetrain, transmission, and car type of the vehicle. A separate Make table was also formed with the model as the primary key and the make attribute, as the make of the vehicle is entirely dependent on the model. The new Model table has a one-to-many relationship with the Car table as many cars may have the same model and year. The Make table has a one-to-many relationship with the Model table since many models are made by the same make, or manufacturer.

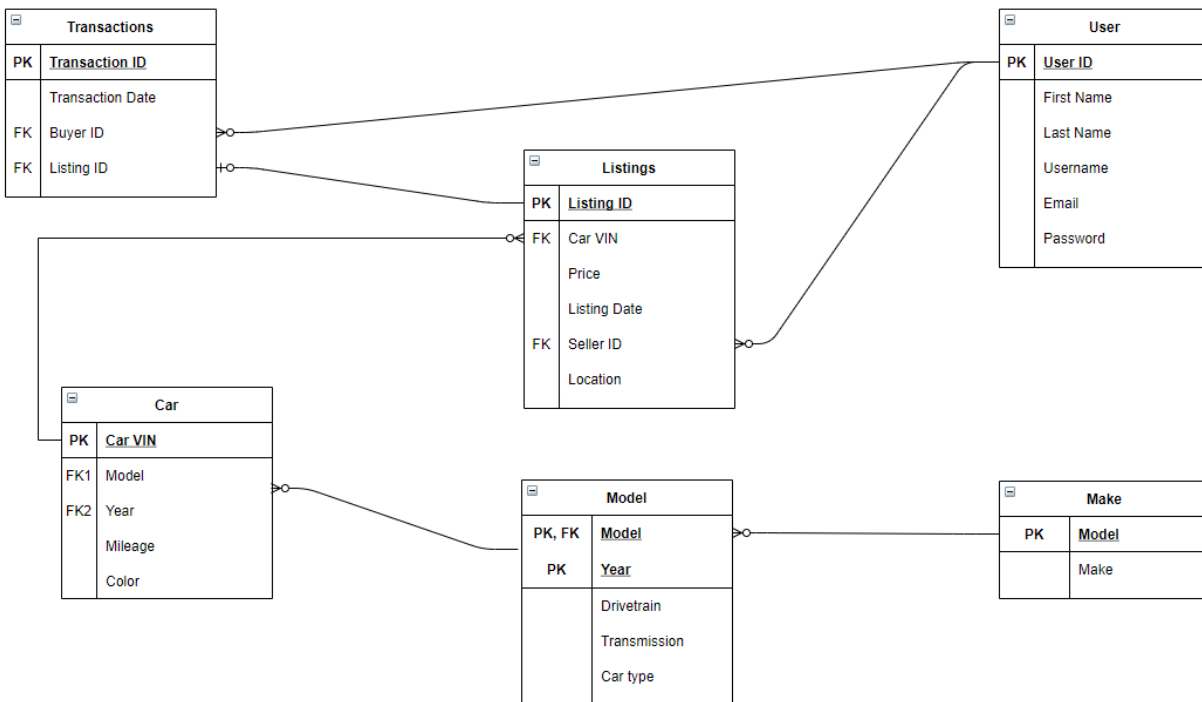
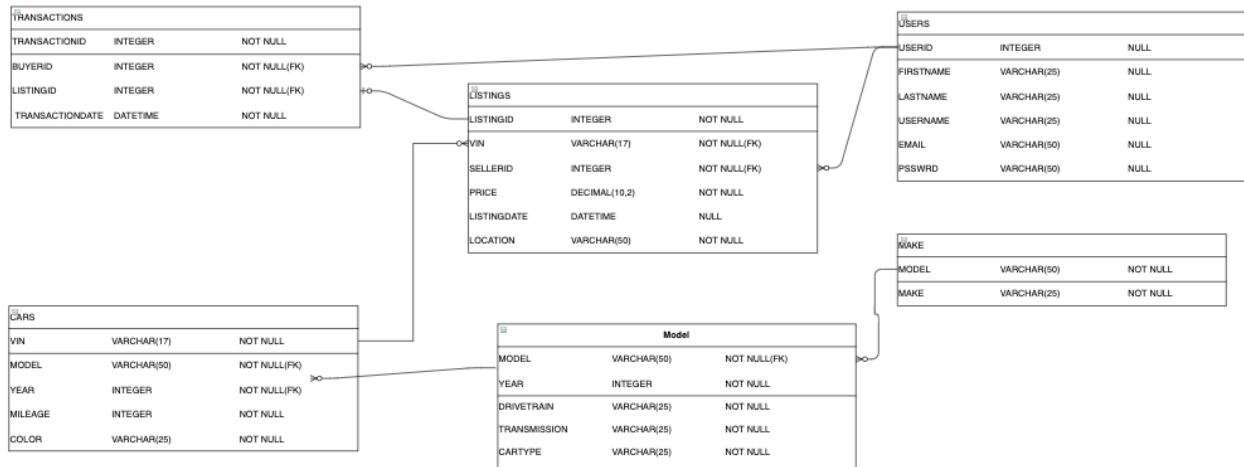


Figure 8 - Third Normal Form

## Physical Design

The physical design of this database was derived from the third normalized form. Tables are converted to show the actual SQL data type along with constraints. In this case, the NULL field constraints are depicted. This is a much more specific ERD that considers necessary items in relation to the DBMS. These data types are all accurate to fit the implementation in the code.



**Figure 9 - Physical Design**

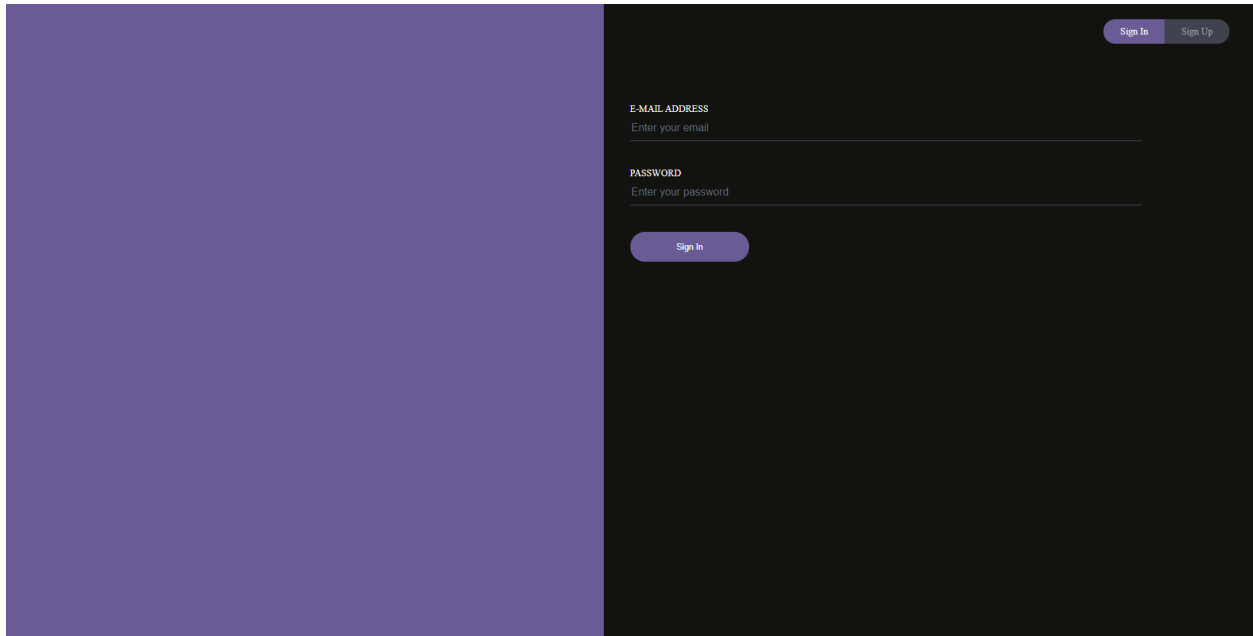
## Implementation and Testing

The implementation for this project was done using a 3-part client-server architecture. The frontend/UI was developed using the React.js framework on the client-side, using help from internal styling and functional libraries from React and React-Dom. In order to enhance application performance, the main structure for routing between pages was done using the React router, providing a significant efficiency advantage for rendering pages over the standard naive html-page approach, as each page relies on functional components that are dynamically rendered in when needed, preventing unnecessary fetches between the client and server for statically linked html pages. The frontend client runs on localhost port 3000.

The backend portion of the application was built using the Express and Node.js frameworks on the server-side. The backend handled the majority of the logic and interaction between the frontend calls and database access and modification. The main structure of the backend consists of several javascript classes, each in charge of a certain base API endpoint corresponding to a table in the database, and one additional class for initializing the database by creating tables with sample data for the first time. Each backend class contained API endpoints for creating, retrieving, updating, and deleting data from tables in the database with various different filters and sub-endpoints. Two additional classes were created for handling user login and registration, and a third class served as the authentication middleware between endpoints for controlling database access to private routes. In order to secure user data for login and database storage, the authentication middleware used the Javascript Web Token (JWT) library to generate and encrypt user passwords in the database, which are later decrypted upon successful user logins and later used to authorize data access to user-specific private routes. In order to communicate with the database, the sqlite3 library was used to pass along get/all/run SQL queries into the database. The backend server runs on port 3001.

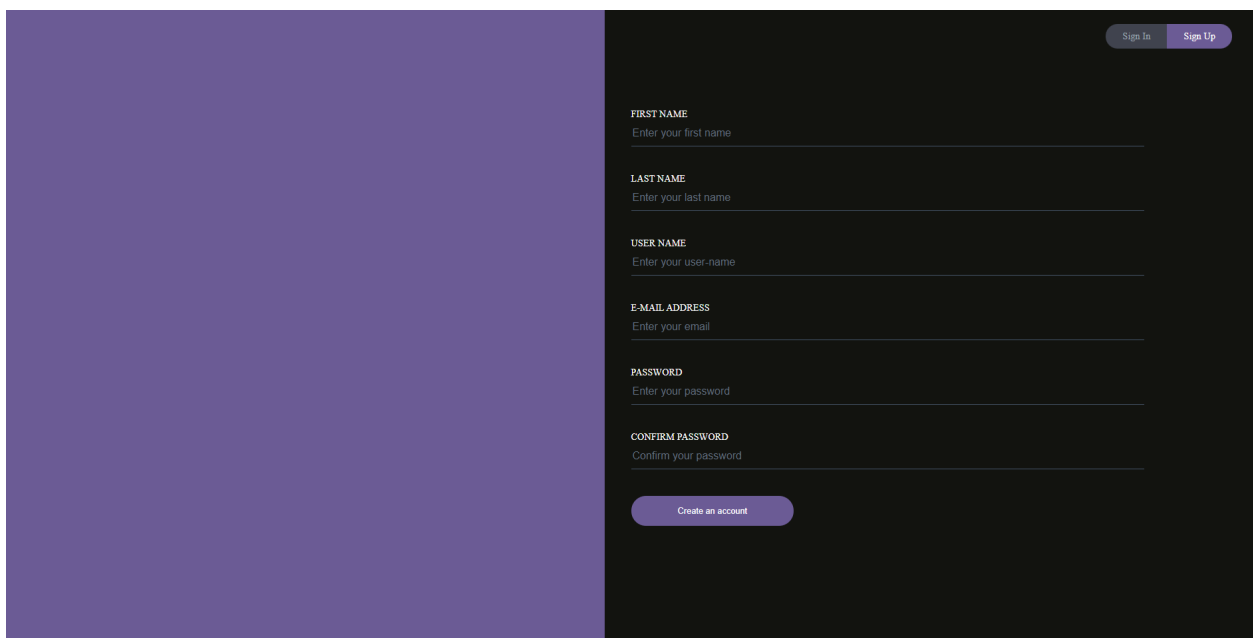
The database portion of the application was built using serverless SQLite test and development databases. Using SQLite provided an additional advantage in terms of portability and flexibility, as it allowed for on-premise storage, modification and secure access to the database locally as compared to a server-based MySQL or Postgres database architecture. As discussed in the backend implementation, communication between the backend and the SQLite database was done using the sqlite3 javascript library. This allowed for successful data creation, modification, access and deletion from the backend to the database. The database can also be locally accessed through the backend server APIs running on port 3001.

Testing separated into two parts: frontend and backend/database testing. During frontend development, testing was done using Nodemon, which made testing very straightforward. Nodemon allowed for instantaneous updating, compiling and running of the most up to date version of the current code during development. This allowed for changes being made to instantly show up on the client side and be tested in real time. Any errors in runtime or compile time showed up on both the client side window as well as the console. Testing for the backend and database sections of the application were done in a different manner using Postman. Postman allowed for individual testing of singular API calls to each endpoint in the backend and provided access to useful information such as HTTP error codes as well as return/response error messages. In correlation to the backend API testing, database testing was also done using SQLite studio and DB browser to check if successful backend queries were being generated or retrieved from the database correctly. The code also maintained two versions of the database, one for testing and one for release. After successful modification and testing of a new feature in the test database, the release database would be updated to reflect the stable changes while new features begin development again on the test database.



The image shows a sign-in page layout. On the left is a large purple rectangle. On the right is a dark gray area containing a sign-in form. At the top right of the dark gray area are two buttons: "Sign In" (active) and "Sign Up". The form has two input fields: "E-MAIL ADDRESS" with the placeholder "Enter your email" and "PASSWORD" with the placeholder "Enter your password". Below these fields is a "Sign In" button.

Figure 10 - Sign In page



The image shows a sign-up page layout. On the left is a large purple rectangle. On the right is a dark gray area containing a sign-up form. At the top right of the dark gray area are two buttons: "Sign In" and "Sign Up" (active). The form has five input fields: "FIRST NAME" (placeholder: "Enter your first name"), "LAST NAME" (placeholder: "Enter your last name"), "USER NAME" (placeholder: "Enter your user-name"), "E-MAIL ADDRESS" (placeholder: "Enter your email"), and "PASSWORD" (placeholder: "Enter your password"). Below the password field is a "CONFIRM PASSWORD" field with the placeholder "Confirm your password". At the bottom of the form is a "Create an account" button.

Figure 11 - Sign Up page

## CAR LISTINGS

Make:

Model:

Mileage at most:

Year newer than:

VIN	MAKE	MODEL	YEAR	LOCATION	MILEAGE	PRICE	
JH4KA7532NC036794	Lexus	is350	2008	San Jose, CA	90000	13000	<input type="button" value="View Listing"/>
YS3AK35E4M5002999	Toyota	Camry	2011	San Jose, CA	120000	10000	<input type="button" value="View Listing"/>
JN1CA31D3YT717809	Audi	A4 Avant	2018	San Jose, CA	40000	14000	<input type="button" value="View Listing"/>
WVGVB75N19W507096	Honda	Accord	2004	San Jose, CA	170000	8300	<input type="button" value="View Listing"/>

Figure 12 - Listing Page UI

## CAR DETAILS:

VIN	MAKE	MODEL	YEAR	PRICE	LISTING DATE	LOCATION	COLOR	MILEAGE	DRIVETRAIN	TRANSMISSION	TYPE	SELLER
JH4KA7532NC036794	Lexus	is350	2008	13000	2021-01-01 10:00:00	San Jose, CA	Black	90000	RWD	Automatic	Sedan	Electrex

Figure 13 - Individual Listing UI

## MY TRANSACTIONS

TRANSACTION ID	LISTING ID	SELLER	TRANSACTION DATE	TRANSACTION AMOUNT
2	4	Electrex	2021-12-08T06:16:12.933Z	8300

Figure 14 - User Transactions UI



## Conclusion

This project sought to apply relational database management principles to a real-world use case, a vehicle resale website. The conceptual design of this project was completed through 4 user views and relevant storyboards with screen flows. This project also explored logical design of the database by normalizing the conceptual user views up to the third normal form. The logical design was carried out by mapping the entities of the logical design to a physical model, with designated data types. Separate SQLite databases were utilized to have both a testing and a development database. A user interface was also developed to facilitate the display of the user views, with a backend to make the SQL commands needed to join those views from the database tables. Dockerfiles and a docker-compose.yaml file are included in the repository of this project to easily deploy the frontend, backend, and database by running the commands “*docker-compose build*” then “*docker-compose up*” from command line. The application can also be deployed from the public Docker images that are available by running “*docker network create project*”, “*docker run -p 3001:3001 -d --network=project --name=node-server electrex/node-server:2.0*”, and “*docker run -p 3000:3000 -d --network=project --name=react-client electrex/react-client:2.0*”.

## Appendix

Project source code: <https://github.com/MerrillPE/cs157a-project>

- Backend: <https://github.com/MerrillPE/cs157a-project/tree/main/backend/routes/api>
- SQL commands used for the database can be found in these classes in the above folder:
  - initdb.js → SQL commands for initializing the database for the first time
  - users.js → SQL commands for interacting with the Users table
  - listings.js → SQL commands for interacting with the Listings table
  - cars.js → SQL commands for interacting with the Cars table
  - makes.js → SQL commands for interacting with the Makes table
  - models.js → SQL commands for interacting with the Models table
  - transactions.js → SQL commands for interacting with the Transactions table
- Additional classes:
  - auth.js → class used for middleware for authentication user token for private routes