Carvana

CST2250 – Software Engineering Management and Development

Middlesex University

Group members

Alexander Ley – Team Leader

Tanush Shah – Secretary

Arda Kecialan M00944158 – Tester

Aqab Javed – Developer

Introduction

Due to increasingly lax antitrust regulations, many consumers have been priced out of markets like car rentals, facing excessive fees and restrictive terms. As a result, digital platforms have become essential for accessing affordable rental options. This project addresses that need by developing a car rental website that allows users to easily browse, compare, and reserve vehicles based on their preferences.

The system focuses on three core areas: a user-friendly search interface, a robust backend to manage listings and accounts, and key features such as filtering and account management to improve user experience.

The project faced early challenges due to changes in team structure, requiring a shift in responsibilities and timelines. Despite this, the team prioritized essential features to deliver a functional and reliable platform.

This report covers the system's design, development, and testing, organized into the following sections:

- Database Setup
- Data Algorithms
- Testing

Each section outlines its role, key implementation details, and related testing.

Database

The database has 3 main structures: Rental Contracts, Customers and Cars. The Car structure contains 2 other structures, namely Model and Rental Status. The Model database stores car models and their details, which are dependent on the car model. Rental Status is an Enum data type that tracks a car's availability.

The Customer database is comprised of a License and personal details tied to the account. Any sensitive information was encrypted using Hash, when pulling data in and out of the database. All details of the customer were available to change accordingly with API endpoints.

The rental contract contains key details about the agreement, including information about the car and the customer involved. Most of these details are dynamically calculated at the time of rental and are not directly editable by the user. For example, the total price cannot be manually modified, but it is automatically updated based on changes to the contract's start and end dates.

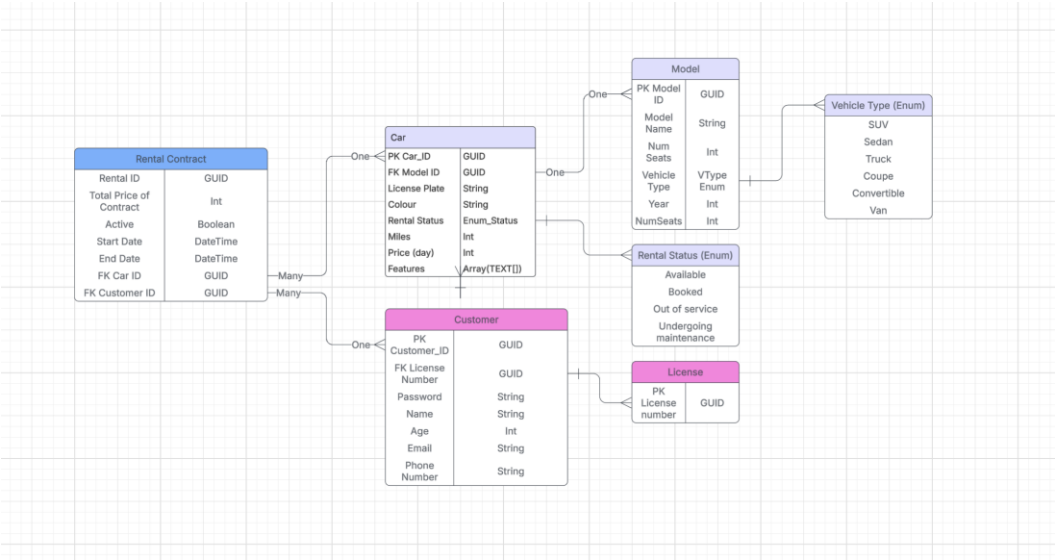

Fig.1

Data Algorithms

The back end is written entirely in C#, and all libraries used are listed at the end of this report. The API has 3 controllers, each handling their own operations independently and asynchronously. These controllers are as follows;

- Car Controller – handles endpoints related to car listings, searches for cars/models based on name or ID, and creating or removing database entries for cars by interacting with Car Service.
- Customer Controller – Handles endpoints tied to user related actions such as logging in, signing up, retrieving customer profiles, and updating customer information by interacting with Customer Service.
- Tree Controller – Handles endpoints related to autocomplete, such as returning suggestions from a prefix, incrementing node weight once a result is selected and startup/debug functions such as pruning and displaying the Trie.

Each of these endpoints comes with error checking, input validation and return types indicating success or failure. In the case of failure, it also returns comments detailing the nature of the problem that occurred.

The main algorithms used in the backend are all related to the Tree Controller, and they encompass tree traversal, pruning, search and autocomplete. Autocomplete uses a Trie implementation, building words using nodes letter by letter and branching to a new one when required. As this is very memory inefficient, we require pruning; the act of reducing the number of nodes in the tree by merging data of multiple nodes which do not have branches and removing the unneeded ones. Shown below (Figure 2) is an example displaying this behaviour. When a node has only one child, the child's data is merged into the parents and is removed from the chain by updating the pointers controlling the structure of the tree. Shown here (Figure 3) is the pseudocode for this action. Retrieving information from this tree is done using a tree traversal to find the node corresponding to the final character in a string and then returning the top 5 outcomes from that point, sorted by their weight. An outcome is defined by a node with attribute "_isFullWord" as true, to indicate that this is a completed phrase that the user could search for. In practice this returns the top 5 outcomes within our system branching from the end of the string the user is attempting to search. Figure 4 shows the pseudocode for these operations. The pseudocode for these was generated by ChatGPT using the groups working C# code for the prompt at the time of writing the report, as we were unable to locate the original pseudocode used during development.

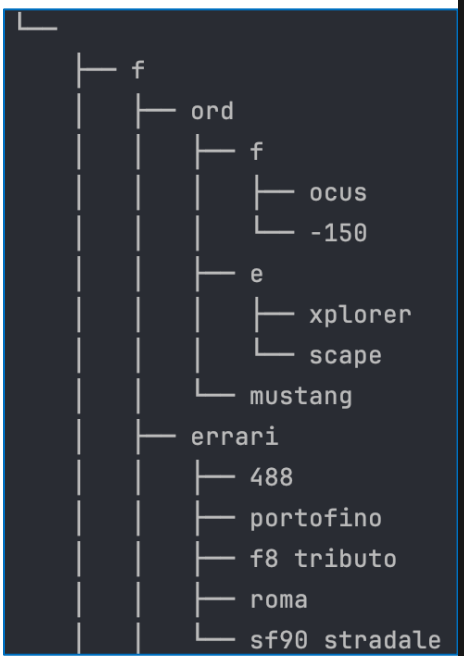

Fig.2

```
FUNCTION PruneNode(currentNode):

    IF currentNode is null:
        Log warning: "Unexpected null node"
        RETURN false

    numberOfChildren ← Count of children of currentNode

    IF numberOfChildren is 0:
        RETURN true

    ELSE IF numberOfChildren is 1:
        child ← the only child of currentNode

        IF currentNode does NOT represent a complete value:
            Merge currentNode with child (call ReplaceNode)

        RETURN PruneNode(child)

    ELSE:
        result ← false
        FOR EACH child in currentNode's children:
            result ← result OR PruneNode(child)

        RETURN result
```

Fig.3

```
FUNCTION AutoComplete(rootNode, prefix):
    IF rootNode is null: RAISE "Missing root node"
    results ← empty list
    suggestions ← GetSuggestionsFromPrefix(rootNode, prefix)
    SORT suggestions by weight descending
    FOR EACH node IN suggestions:
        ExpandToFullWords(node, "", results)
    RETURN first 5 items in results

FUNCTION ExpandToFullWords(node, current, list):
    current ← current + node.data
    IF node is full word: ADD current to list
    FOR EACH child IN node.children:
        ExpandToFullWords(child, current, list)

FUNCTION GetSuggestionsFromPrefix(rootNode, prefix):
    current ← rootNode
    WHILE prefix not empty:
        match ← false
        FOR EACH child IN current.children:
            IF prefix starts with child.data:
                REMOVE matched part from prefix
                current ← child
                match ← true
                BREAK
        IF not match: RETURN null
    RETURN all descendant nodes that form full words
```

Fig.4

Testing Table

Test ID	Description	Expected Result	Actual Result	Status
T1	Verify tree loads correctly from file	Root node created, words inserted	As expected	Pass
T2	Ensure loading handles empty files gracefully	Root exists, but no children	As expected	Pass
T3	Check that words are correctly added to the tree	Correct node structure forms	As expected	Pass
T4	Return correct suggestions for a prefix	Returns ["Toyota", "Tesla"]	As expected	Pass
T5	Ensure no results for an invalid prefix	Returns empty list	As expected	Pass
T6	Ensure pruning reduces redundancy	Tree size reduces, words preserved	As expected	Pass
T7	Ensure tree prints correctly	Properly formatted output	As expected	Pass
T8	Ensure API returns correct matches for a prefix	Returns ["Toyota", "Tesla"]	As expected	Pass
T9	Handle case where input is empty	Returns HTTP 400 Bad Request	As expected	Pass
T11	Database Connection Test	No errors, successful connection	As expected	Pass
T14	localhost:5046/auth/signup	Account created successfully	As expected	Pass
T15	localhost:5046/auth/signup (with same id)	Account already exists	As expected	Pass
T16	localhost:5046/auth/login?email =TTT@gmail.com&password=Test22	TTT@gmail.com	As expected	Pass
T17	localhost:5046/auth/login?email =TTT@gmail.com&password=AAAA	Invalid credentials	As expected	Pass
T18	localhost:5046/auth/profile (UPDATE)	Users detail should be updated	As expected	Pass
T21	localhost:5046/rent/models/search/tesla	Information for tesla	As expected	Pass
T24	localhost:5046/rent/count	Rented cars	As expected	Pass
T25	localhost:5046/search/initialise	Results outputted to console	As expected	Pass
T26	localhost:5046/search?prefix=a	All the cars brands starts with a display without a	As expected	Pass

T28	Database Checks	Make sure all the details matches	As expected	Pass
T29	Encryption Checks	Encrypted password on db	As expected	Pass
T30	User registration functionality on the website.	successful when data is correctly formatted. If the format is incorrect, a warning is displayed and the details are not stored in the database.	As expected	Pass
T33	Profile page of the website.	The correct information should be displayed	As expected	Pass
T35	Update page	View the updated details.	All credentials disappeared; had to log out and log back in to see updated details.	Fail Fix and retest required
T38	Rent page	View the cars; when a filter is applied, cars that don't match should disappear from the page	As expected	Pass
T40	Log-out function	Account should be logged out, redirected to the home page	As expected	Pass
T41	Autocomplete function	Should suggest only brand names that start with the user's input. If partially typed, only matching brands should appear.	As expected	Pass

Conclusion

This project has successfully created a solid framework for a car rental site, encompassing a functional database hosted on AWS servers for cars and customers, working autocomplete and filter features to allow users to find the product more easily they are looking for. The project was heavily limited by situations outside of the groups control, such as one member leaving the university and another having scheduling issues mostly stopping him from being able to contribute. Due to this the team of 3 had to cut back on many of the intended features and focus on polishing the core requirements.

If this project were attempted again in the future some core oversights would have been handles differently, such as encrypting sensitive user information such as passwords and emails with AES as it is non-deterministic and should not be used over hashing.

Overall, the rest of the project was well planned and with fewer external setbacks it would have been able to achieve all its objectives.

References

Comments have been placed within the code to cite sources and show when ideas were taken.

Libraries used:

- Microsoft EFCore
- Microsoft AspNetCore
- System.ComponentModel.DataAnnotations
- System.Text.Json.Serialization
- System.Security.Cryptography