CarCraftr

Start of the self-employment: 15th August

What is CarCraftr?

Carcraftr is a web and iOS app that enables users to compare cars. The app focuses on letting users compare various car models based on features like safety, performance, and pricing. It also offers features like:

- Viewing upcoming car models
- Current offers and deals
- Car comparison based on specifications (like fuel efficiency, engine power, etc.)
- Potentially adding a future feature for estimating prices of used cars

Carcraftr combines car research and decision-making elements, aiming to make selecting and comparing cars easier for users.

Techstack that will be used to develop CarCraftr:

For Carcraftr, a robust tech stack is essential to handle the front-end user interface and the back-end logic, including APIs, databases, and server management. Here's a recommended tech stack:

Front-end:

1. React (JavaScript/TypeScript):
    - Build a dynamic, responsive web app interface.
    - Handle car comparison UI and manage state with React Hooks.
    - Use React Router for seamless navigation between pages (Home, Compare, Car Details, etc.).
2. Redux (Optional for State Management):
    - Manage application state more effectively, especially when dealing with complex car data comparisons.
3. HTML5 & CSS3:
    - Structure the UI and design components using CSS for a clean, user-friendly design.
    - Consider using CSS Frameworks like Bootstrap or Material-UI for pre-built responsive design components.
4. Axios/Fetch API:
    - To handle API calls from the backend and fetch car data (from the Edmunds API or your API).
5. Swift (for iOS App Development):
    - Use Swift and SwiftUI for building the iOS app version of Carcraftr.
    - Integrate the backend API and handle user interactions.

Back-end:

1. Node.js with Express.js:
    - Handle server-side logic for the app.
    - Create RESTful API endpoints for fetching and comparing car data.
    - Manage user authentication, if necessary.
2. MongoDB (NoSQL Database):
    - Store car details, user preferences, and comparison history.
    - Use Mongoose as an ORM for modeling car and user data.
    - Scalable and efficient for large datasets, such as car details.
3. Edmunds API (or another car data API):
    - Fetch car data, specifications, pricing, and other details from Edmunds.
    - Process API responses to provide users with real-time information.

DevOps/Infrastructure:

1. Docker (Optional):
    - Containerize the application to make it easier to deploy and manage across environments.
2. Heroku or AWS (Hosting Platform):
    - Host the Node.js/Express backend on a cloud provider for scalability and performance.
    - Use S3 for storing static assets (like images or files).
3. CI/CD Tools (e.g., GitHub Actions):
    - Automate testing, building, and deployment processes.

Others:

1. OAuth or JWT for Authentication:
    - Secure user accounts and log in using token-based authentication if user profiles are required.
2. Postman:
    - Test your API during development.
3. Figma or Sketch (for Design):
    - Prototype and design the UI before starting development.

Tech Stack Summary:

- Front-end: React (JavaScript/TypeScript), Redux, Swift (for iOS), HTML5, CSS3, Axios/Fetch

- Back-end: Node.js, Express.js, MongoDB, Edmunds API
- Infrastructure: Docker, Heroku/AWS, GitHub Actions
- Others: OAuth/JWT, Postman, Figma

This stack is flexible and scalable, supporting web and mobile development and allowing for future expansion of Carcraftr.

Brainstormed the idea and came up with epics/stories to complete CarCraftr:

Epic 1: Car Comparison

This epic focuses on the app's core functionality — allowing users to compare car models based on various specifications.

User Stories:

- As a user, I want to search for car models to quickly find the cars I am interested in comparing.
- As a user, I want to select multiple cars to compare so that I can evaluate the differences between them.
- As a user, I want to view detailed car specifications, such as fuel efficiency, engine power, and safety ratings, to make an informed decision.
- As a user, I want to filter cars based on price, brand, or type to narrow my comparison.
- As a user, I want to visually compare selected car features to identify which car best suits my needs quickly.

Epic 2: Upcoming Cars and Offers

This epic involves adding features to show users upcoming car models and current offers.

User Stories:

- As a user, I want to view upcoming car models and consider waiting for a new one before purchasing.
- As a user, I want to see offers and deals on cars so I can take advantage of discounts when deciding on a purchase.
- As a user, I want to receive notifications about new car models or deals to stay informed.

Epic 3: User Accounts and Preferences

This epic enables users to create accounts, save comparisons, and set preferences.

User Stories:

- As a user, I want to create an account to save my car comparisons and preferences.
- As a user, I want to log in to my account to access my saved data from different devices.

- As a user, I want to save my favorite cars or comparisons to return to them later quickly.
- As a user, I want to set preferences for notifications about new cars and offers so I only receive relevant updates.

Epic 4: UI/UX and Design

This epic involves creating a clean and user-friendly design for the web and iOS apps.

User Stories:

- As a user, I want a responsive and intuitive interface to navigate the app easily.
- As a user, I want the design to be visually appealing, so I enjoy using the app.
- As a user, I want a mobile-friendly layout to compare cars on my smartphone easily.

Epic 5: Backend Development & API Integration

This epic is about setting up the server-side logic and integrating APIs like the Edmunds API.

User Stories:

- As a developer, I want to create RESTful API endpoints so the front end can fetch car data and details.
- As a developer, I want to integrate the Edmunds API (can be other) to fetch car specifications, pricing, and deals.
- As a developer, I want to set up a MongoDB database to store user preferences, comparison history, and car data.
- As a developer, I want to implement security measures like JWT for user authentication to protect users' accounts.

Epic 6: Testing and Deployment

This epic focuses on testing the app to ensure functionality and deploying it to a live environment.

User Stories:

- As a developer, I want to create unit tests for API endpoints to ensure they work correctly.
- As a developer, I want to test the front end thoroughly so the user experience is smooth and bug-free.
- As a developer, I want to deploy the app on a platform like Heroku or AWS so users can access it online.

Epic 7: Performance and Scalability

This epic is related to ensuring the app can handle large amounts of data and users efficiently.

User Stories:

- As a developer, I want the app to handle many users simultaneously without slowing down to smooth the experience.
- As a developer, I want to optimize database queries for fast loading times when retrieving car data.
- As a developer, I want to implement caching mechanisms for frequently accessed data to reduce load times and server requests.

Epic 8: Security and Privacy

This epic ensures that user data is protected and secure.

User Stories:

- As a user, I want my personal information (email, saved cars, etc.) to be kept private and secure.
- As a user, I want to be able to delete my data if I no longer wish to use the app.
- As a developer, I want to secure the API and user data with encryption and authentication to protect sensitive information.

Epic 9: Analytics and Insights

This epic gathers user data to provide insights into car trends and preferences.

User Stories:

- As a user, I want to see the most popular car models being compared by other users to discover trending cars.
- As a user, I want to receive personalized car suggestions based on my previous searches and comparisons.
- As an admin, I want to track user behavior (e.g., most compared cars, time spent on comparison) to improve user experience.

Epic 10: Notifications and Reminders

This epic is about informing users through notifications like price drops or new car arrivals.

User Stories:

- As a user, I want notifications about price drops for cars I have saved or compared.
- As a user, I want to get reminders when new car models are released, or special deals are available for cars I'm interested in.
- As a user, I want to receive push notifications on my mobile device to stay updated about new cars and offers.

Epic 11: Future Feature – Admin Management

This epic focuses on allowing admins to manage the app's content, such as adding new car data or managing user accounts.

User Stories:

- As an admin, I want to add, update, and remove car models from the database so that the app stays up-to-date with the latest cars.
- As an admin, I want to manage user accounts (e.g., reset passwords and deactivate accounts) to maintain control over the user base.
- As an admin, I want to view app usage statistics (e.g., popular car models and user activity) to improve the app's functionality.

Epic 12: Future Feature – Used Car Marketplace

This epic involves adding a platform for users to buy and sell used cars with estimated pricing.

User Stories:

- As a user, I want to browse listings for used cars to find affordable options.
- As a user, I want to see estimated prices for used cars to evaluate their value.
- As a seller, I want to list my car for sale to reach potential buyers.
- As a buyer, I want to filter used cars by price and condition to find a vehicle that meets my budget and requirements.

CRUD operations for creating API's:

CRUD (Create, Read, Update, Delete) operations for managing car data through a RESTful API. It includes instructions for creating a UI form that interacts with the backend API to handle car entries.

Setting Up the Backend API

To begin, initialize your Node.js project and install the necessary packages. You can do this by running npm init -y followed by npm install express mongoose body-parser cors. Next, set up a basic Express server by creating a new JavaScript file and including the required libraries. The server should connect to a MongoDB database and handle JSON requests.

Create an Express server with the following code snippet:

```
const express = require('express');
const mongoose = require('mongoose');
const bodyParser = require('body-parser');
const cors = require('cors');
```

```
const app = express();
const PORT = process.env.PORT || 5000;

mongoose.connect('mongodb://localhost:27017/carcraftr', {
  useNewUrlParser: true,
  useUnifiedTopology: true
});

app.use(cors());
app.use(bodyParser.json());
```

Define a Mongoose schema for the car data. The following code defines a simple schema that includes fields such as name, brand, year, and price:

```
const CarSchema = new mongoose.Schema({
  name: String,
  brand: String,
  year: Number,
  price: Number
});

const Car = mongoose.model('Car', CarSchema);
```

Once a model is established, we can create the necessary CRUD API endpoints. This includes a POST endpoint to create a new car, a GET endpoint to retrieve all cars, a PUT endpoint to update an existing car, and a DELETE endpoint to remove a car.

```
// Create a new car
app.post('/api/cars', async (req, res) => {
  const newCar = new Car(req.body);
  await newCar.save();
  res.status(201).json(newCar);
});

// Get all cars
app.get('/api/cars', async (req, res) => {
  const cars = await Car.find();
  res.json(cars);
});
```

```
// Update a car
app.put('/api/cars/:id', async (req, res) => {
    const updatedCar = await Car.findByIdAndUpdate(req.params.id, req.body, { new: true });
    res.json(updatedCar);
});

// Delete a car
app.delete('/api/cars/:id', async (req, res) => {
    await Car.findByIdAndDelete(req.params.id);
    res.status(204).send();
});

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

Frontend Integration

To connect the front end with the back end, you need to create a form component in React to handle user input for car data. This component will use Axios to send data to the API.

```
import React, { useState } from 'react';
import axios from 'axios';

function CarForm() {
    const [car, setCar] = useState({ name: '', brand: '', year: '', price: '' });

    const handleChange = (e) => {
        const { name, value } = e.target;
        setCar({ ...car, [name]: value });
    };

    const handleSubmit = async (e) => {
        e.preventDefault();
        await axios.post('/api/cars', car);
        // Optionally reset the form
        setCar({ name: '', brand: '', year: '', price: '' });
    };
```

```
    return (
        <form onSubmit={handleSubmit}>
            <input name="name" placeholder="Car Name" onChange={handleChange} />
            <input name="brand" placeholder="Brand" onChange={handleChange} />
            <input name="year" type="number" placeholder="Year" onChange={handleChange} />
            <input name="price" type="number" placeholder="Price" onChange={handleChange} />
            <button type="submit">Add Car</button>
        </form>
    );
}

export default CarForm;
```

In this component, the handleChange function updates the state whenever the user inputs data. The handleSubmit function sends a POST request to create a new car entry in the database upon form submission.

PS: This is a basic structure and can be expanded further with future developments