Car Rental-Commerce Platform Report
Hakathon 3 | Day 3
Name: Muhammad Abubakar
Roll Number: 00420397
Class: Friday (7 PM to 10 PM)

Overview

The primary focus of Day 3 was integrating APIs with **Sanity CMS**, utilizing **GROQ queries**, and updating the schema to meet dynamic data requirements. The day's activities were aimed at enhancing functionality and ensuring seamless data transfer between the platform's frontend and backend.

---

1. API Integration Process

Process

1. **Identify Required APIs**: Defined the APIs needed for car-related data.
2. **Configure API Keys and Endpoints**: Ensured secure configurations for API access.
3. **HTTP Requests Setup**: Established Fetch-based mechanisms for efficient data transfer.
4. **Error Handling**: Implemented robust mechanisms for API call reliability.

Steps Taken

- Integrated APIs for fetching **car listings**.
- Secured API requests using authentication tokens.
- Debugged and tested API functionality through **Postman**.

2. Adjustments to Schemas

Conceptual Comparison of Old and New Schemas

Schemas in a system serve as blueprints for data representation. The **old schema** was designed with a focus on basic data fields to establish initial functionality. However, the **new schema** introduces enhancements to improve usability, scalability, and API integration compatibility.

Old Schema

The old schema was structured for simplicity and contained:

1. **Car ID**: Unique identifier.
2. **Brand**: Car's brand.
3. **Model**: Model name or number.

4. **Year**: Manufacturing year.
5. **Category**: Classification (e.g., SUV, Sedan).
6. **Pricing**: Cost or rental price.
7. **Availability**: Rental availability status.
8. **Features**: Additional details (e.g., GPS, Bluetooth).

New Schema

The new schema expands the capabilities by introducing user-friendly labels, additional fields, and a structure optimized for API integrations:

```
export default {
  name: 'car',
  type: 'document',
  title: 'Car',
  fields: [
    {
      name: 'name',
      type: 'string',
      title: 'Car Name',
    },
    {
      name: 'brand',
      type: 'string',
      title: 'Brand',
      description: 'Brand of the car (e.g., Nissan, Tesla, etc.)',
    },
    {
      name: 'type',
      type: 'string',
      title: 'Car Type',
      description: 'Type of the car (e.g., Sport, Sedan, SUV, etc.)',
    },
    {
      name: 'fuelCapacity',
      type: 'string',
      title: 'Fuel Capacity',
      description: 'Fuel capacity or battery capacity (e.g., 90L, 100kWh)',
    },
    {
      name: 'transmission',
      type: 'string',
      title: 'Transmission',
      description: 'Type of transmission (e.g., Manual, Automatic)',
    },
    {
      name: 'seatingCapacity',
      type: 'string',
      title: 'Seating Capacity',
      description: 'Number of seats (e.g., 2 People, 4 seats)',
    },
    {
      name: 'pricePerDay',
      type: 'string',
      title: 'Price Per Day',
      description: 'Rental price per day',
    },
    {
```

```
    name: 'originalPrice',
    type: 'string',
    title: 'Original Price',
    description: 'Original price before discount (if applicable)',
   },
   {
    name: 'tags',
    type: 'array',
    title: 'Tags',
    of: [{ type: 'string' }],
    options: {
     layout: 'tags',
    },
    description: 'Tags for categorization (e.g., popular, recommended)',
   },
   {
    name: 'image',
    type: 'image',
    title: 'Car Image',
    options: {
     hotspot: true,
    },
   },
  ],
};
```

Key Updates in the New Schema

1. **Car Name**: Combines brand and model for better identification.
2. **Brand**: Retained for brand-specific filtering.
3. **Car Type**: Simplifies categorization using user-friendly labels like "Sport" or "Luxury."
4. **Fuel Capacity**: Highlights fuel or battery specifications.
5. **Transmission**: Specifies gearbox type (Manual/Automatic).
6. **Seating Capacity**: Indicates the number of seats.
7. **Price Per Day**: Added for rental calculations.
8. **Tags**: Enables better car categorization (e.g., "popular," "luxury").
9. **Image**: Allows for attaching images for frontend visualization.

3.Data Migration with Sanity CMS:

1. Setting Up Sanity CMS:

- **Library Installation**: Installed the @sanity/client library to easily connect with Sanity's API.

2. Setting Up Sanity Client:

- **Configuration**: Set up the Sanity client using important project details like Project ID, Dataset, and API Version, making sure the connection to Sanity is secure and works smoothly.

3. Creating Custom Schemas:

- **Car Details Schema**: Designed a special schema to store all the car information. This includes:
    - Car Name

- Car Type
- Price Per Day
- Availability
- Image URL

4. Fetching and Displaying Data:

- **API Fetch**: Used the Sanity API to fetch data, so the platform always shows the latest details about cars, customers, and bookings.
- **Real-Time Updates**: The platform automatically updates when there are changes in Sanity CMS, such as changes in car availability, prices, or bookings. This keeps everything up-to-date without having to refresh manually.

5. Frontend Integration:

- **Displaying Data**: The fetched data is displayed on the platform's frontend, ensuring users can easily browse cars, check customer records, and manage bookings.

Why Sanity CMS is Ideal:

1. Centralized Data Management:

- All the information is managed in one place, making it easier to handle and update.

2. Real-Time Sync:

- Any changes to car availability, pricing, or bookings are instantly reflected on the platform, keeping everything current.

3. Flexible Content Models:

- Sanity lets you create custom schemas that fit the needs of a car rental platform, ensuring the data is well-organized.

4. Fast Performance:

- Sanity's built-in features, like image optimization, make sure that car images load quickly, improving the platform's speed.

5. Scalable for Growth:

- As the platform grows, Sanity CMS can handle more data and more users, ensuring everything runs smoothly.

4. Screen Shots of working:

1. Api calling :

API calling is the process of sending requests to a server to fetch or send data. It allows your app to interact with external services or databases to retrieve or update information.

API

```
const response = await fetch("https://sanity-nextjs-application.vercel.app/api/hackathon/template7");

const data = await response.json();
console.log(data)
```

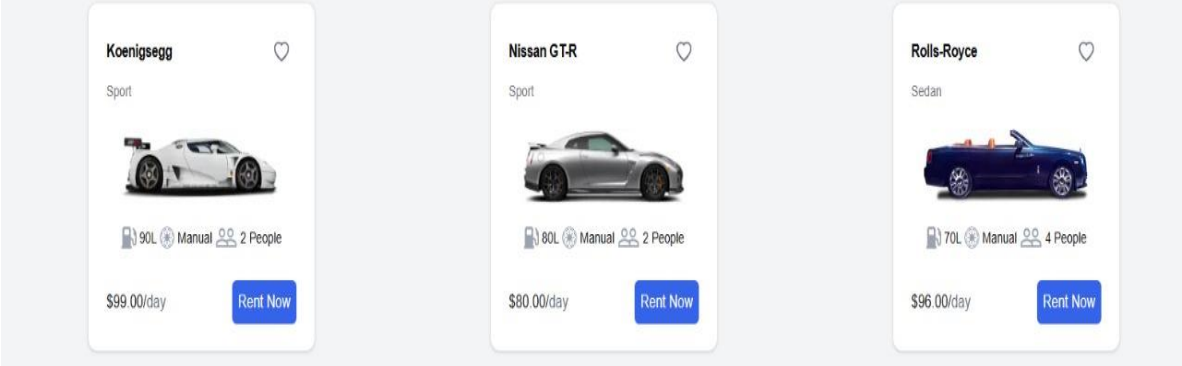
Data with API

```
<div className="grid grid-cols-1 sm:grid-cols-2 lg:grid-cols-3 gap-4 mt-10">
  {data.map((car:Car, index :number) => (
    <Card key={car.id} className="w-full max-w-[304px] mx-auto h-auto flex flex-col justify-between">
      <CardHeader>
        <CardTitle className="w-full flex items-center justify-between">
          {car.name}
          <LikeButton/>

        </CardTitle>
        <CardDescription>{car.type}</CardDescription>
      </CardHeader>
      <CardContent className="w-full flex flex-col items-center justify-center gap-4">
        <img src={car.image_url} alt={car.name} width={220} height={68} />
        <div className="flex items-center space-x-1">
          <FontAwesomeIcon icon = {faGasPump} className="text-gray-400" style={{ width: '20px', height: '20px' }} />
          <span className="text-sm">{car.fuel_capacity}</span>
          <FontAwesomeIcon icon={faGalacticRepublic} className="text-gray-400" style={{ width: '20px', height: '20px' }} />
          <span className="text-sm">{car.transmission}</span>
          <MdPeopleOutline size={30} className="text-gray-400" />
          <span className="text-sm flex">{car.seating_capacity}</span>
        </div>
      </CardContent>
      <CardFooter className="w-full flex items-center justify-between">
        <p>
          {car.price_per_day}/<span className="text-gray-500">day</span>
        </p>
        <button className="bg-[#3563e9] p-2 text-white rounded-md">
          <a href={`/morecars/${car.id}`}>Rent Now</a>
        </button>
      </CardFooter>
    </Card>
  ))}
</div>
```

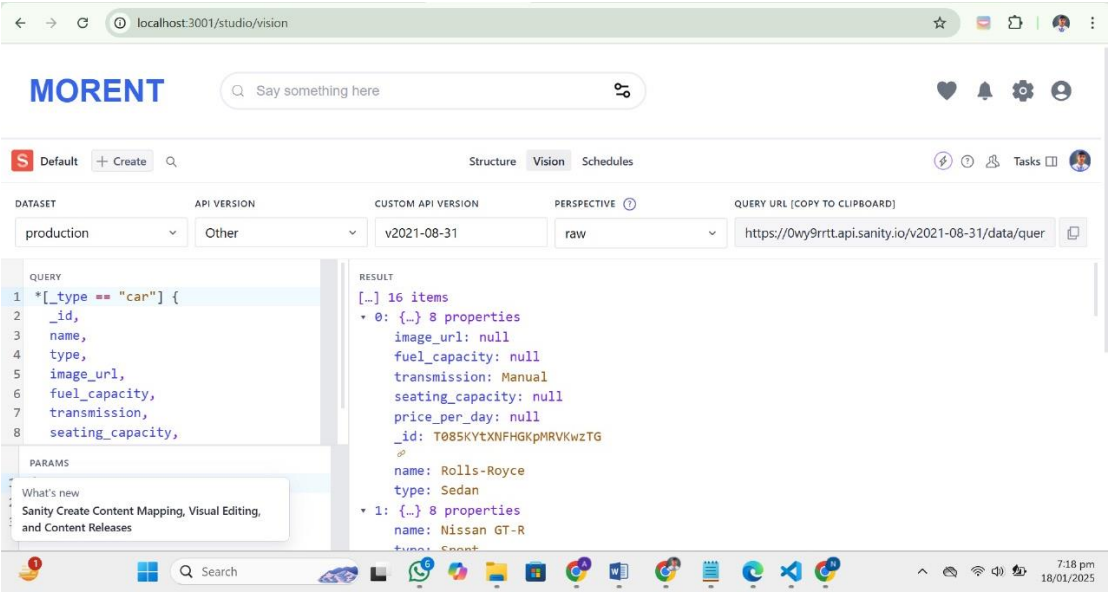
2. Data successfully displayed in the frontend :

Means the fetched data is correctly shown on the website or app for users to view.



3. Populated Sanity CMS fields :

Populated Sanity CMS Fields refers to filling in the necessary data within the Sanity CMS. This ensures that the content, like car details or customer information, is stored and ready to be fetched for display.



Day 3 Checklist:

1. Self-Validation Checklist:

- **API Understanding**: ✅Completed
- **Schema Validation**: ✅Completed
- **Data Migration**: ✅Completed
- **API Integration in Next.js**: ✅Completed
- **Submission Preparation**: ✅Completed

“Today's work involved mastering API calls, ensuring data is correctly populated in Sanity CMS, and successfully displaying it on the frontend. With all tasks checked off, the project is on track and ready for the next steps.”

M a d e b y A b u b a k a r