API Integration and Data Migration Report

Presented By: Shoaib Tahir

This documentation explains the work done on Day 3 of the Rental E-commerce Marketplace hackathon. It includes custom database setup, integrating data from Sanity, creating schemas, and showing data using GROQ queries in a Next.js app. Each part is explained simply with details about how the code works.

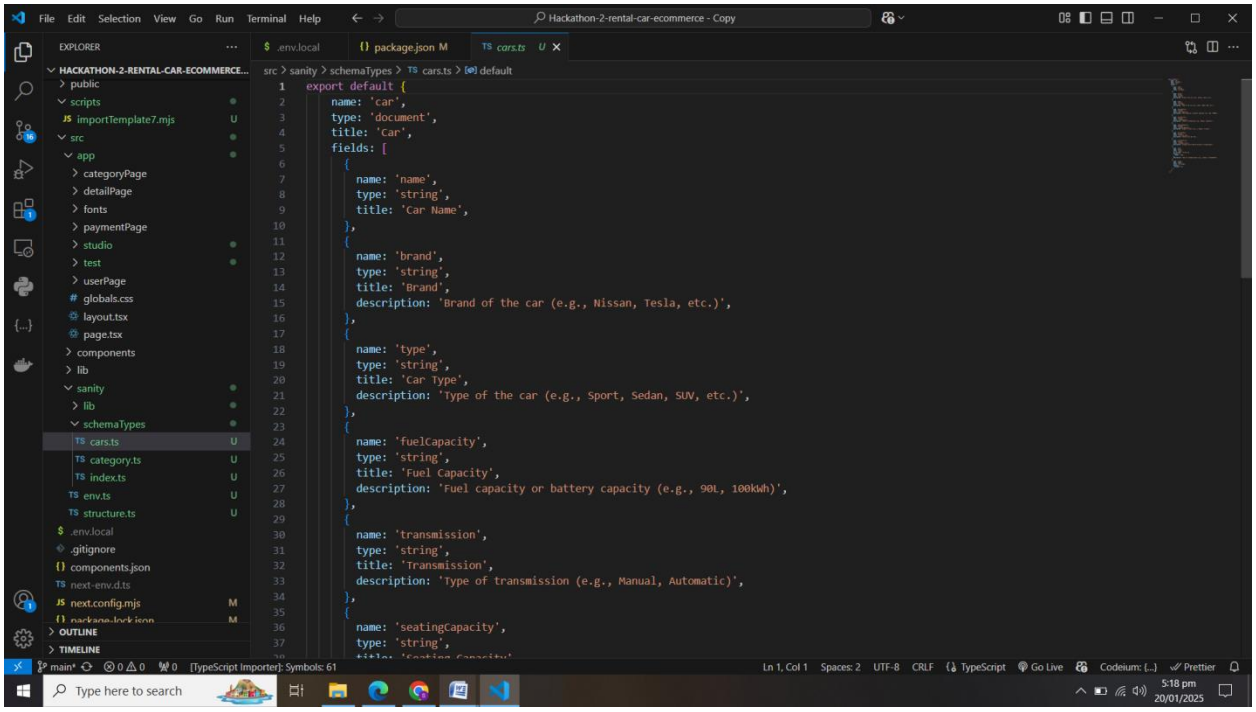
1. Sanity CMS Schema Design:

To ensure the smooth handling of car data, I designed a schema called **car** in Sanity CMS. The schema includes the following fields:

- **Rental Product Schema Fields:**

- **Name:** The name of the product (text).
- **Brand:** The brand of the product (text).
- **Type:** The type or category of the product (text).
- **Fuel Capacity:** The fuel capacity of the product (number).
- **Transmission:** The transmission type (e.g., manual or automatic) (text).
- **Seating Capacity:** The number of people the product can seat (number).
- **Price Per Day:** The rental price per day (number).
- **Original Price:** The original price of the product (number).
- **Tags:** A list of tags for categorization (list of text).
- **Image:** The product's image (image).

Code Snap:



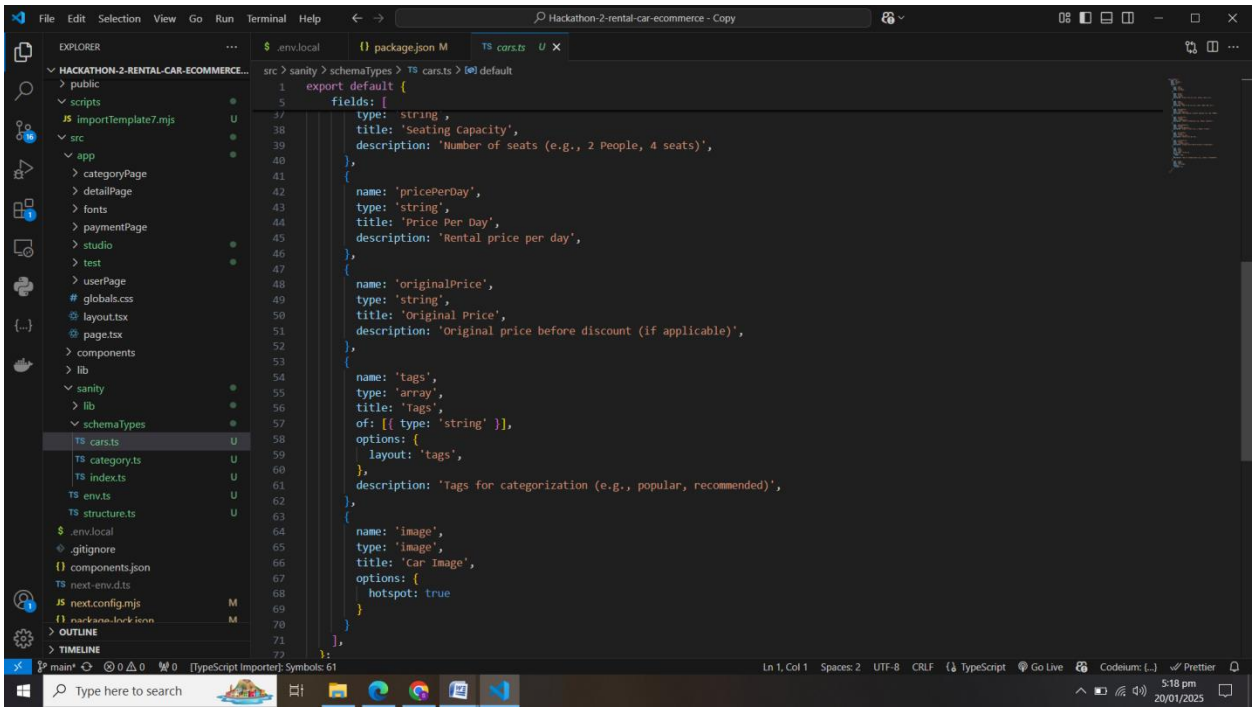
2. API Integration and Data Migration:

- **API Data Fetching:**

I retrieved product data from an external API, including:

- Name
- Brand
- Type
- Fuel Capacity
- Transmission
- Seating Capacity
- Price Per Day
- Original Price
- Tags
- Image

This data was mapped directly to the Sanity CMS schema.

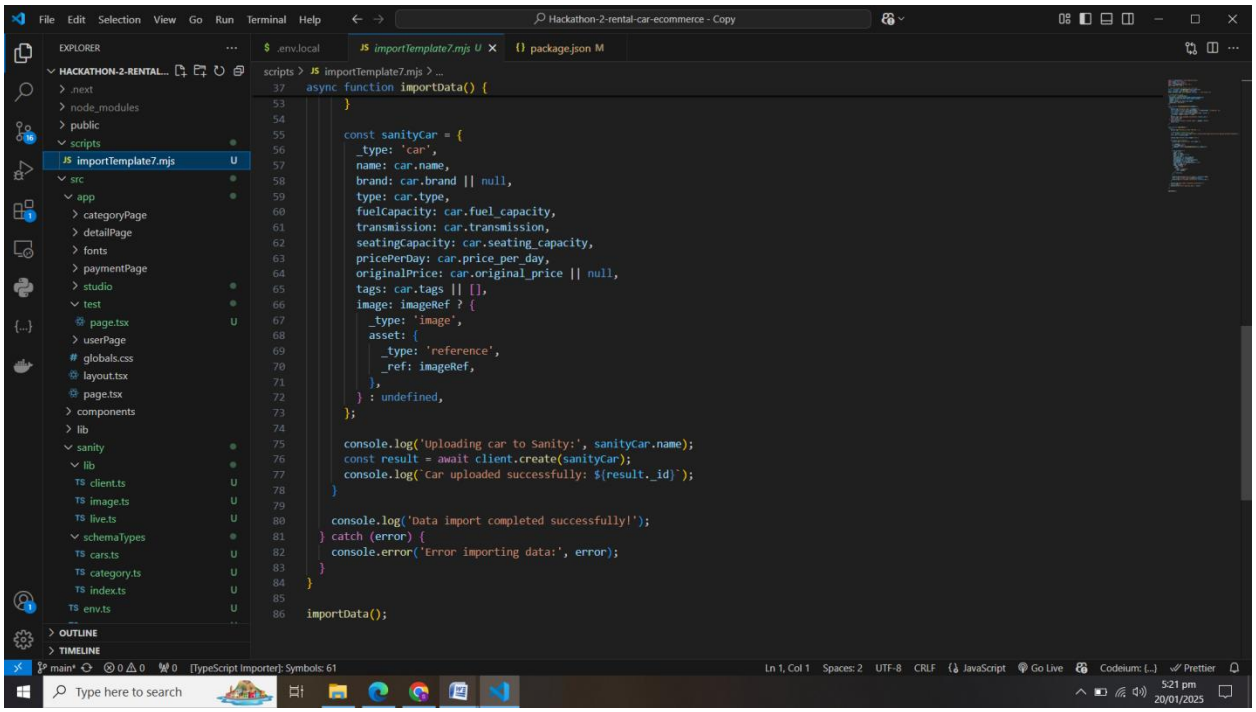


- **Data Population in Sanity CMS:**

After fetching data from the API, I filled the car fields in Sanity CMS automatically. This made sure the car information was consistent and accurate across the platform.

- **Data Migration:**

Using the Sanity CLI, I backed up the dataset from Sanity CMS and later imported it again for testing. This ensured all data was well-organized and displayed correctly on the frontend.



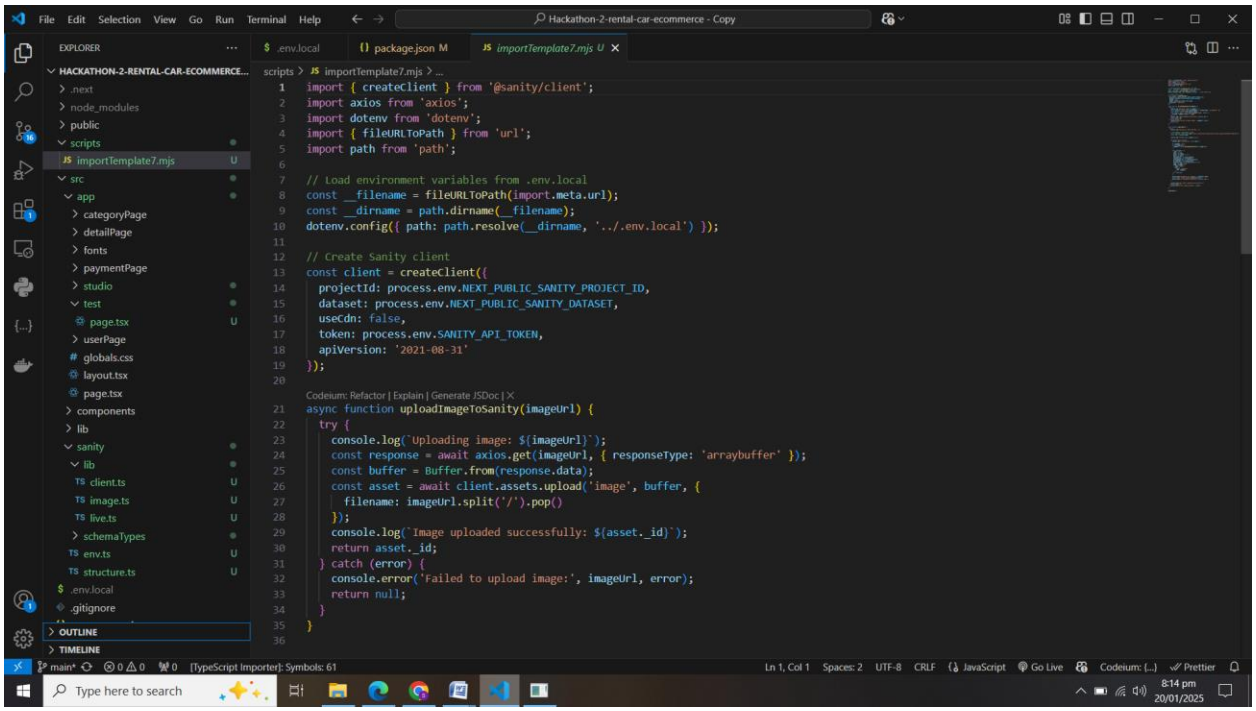
3. Steps Taken for Data Migration:

- **Exporting Data:**

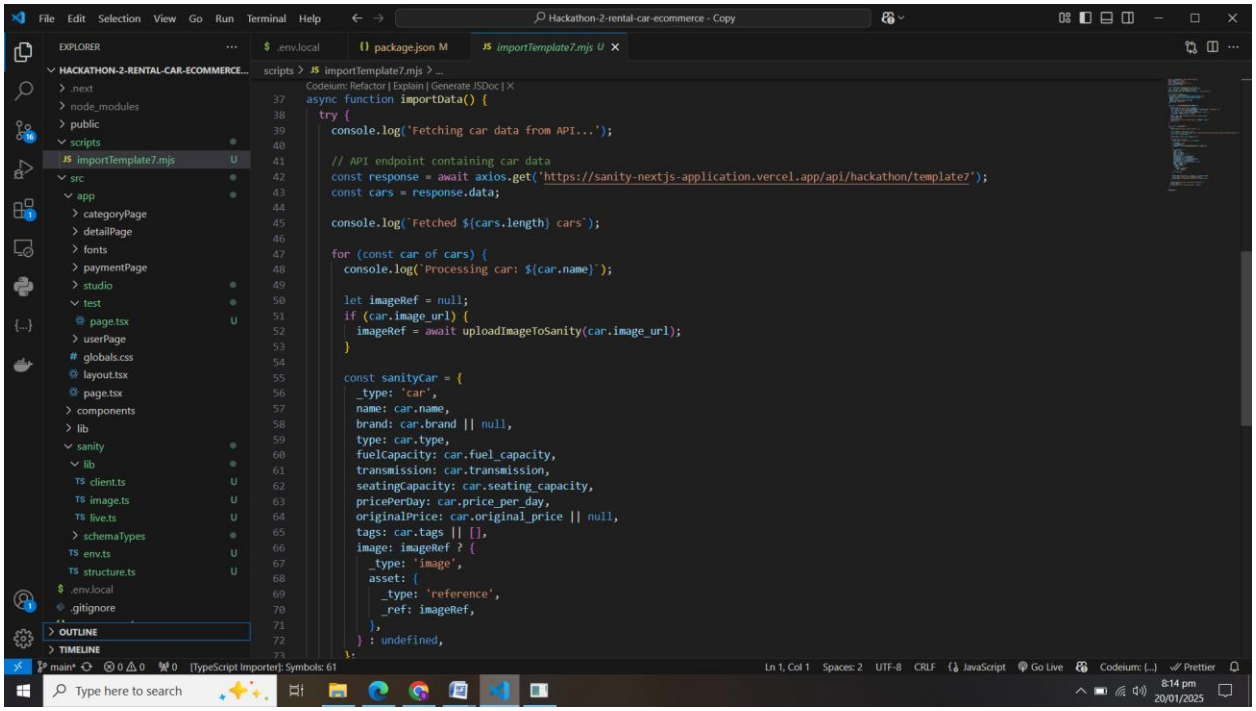
The first step was to export the data from Sanity CMS using the Sanity CLI. This made sure all the car data was safely backed up before doing anything else.

- **Verification of Data:**

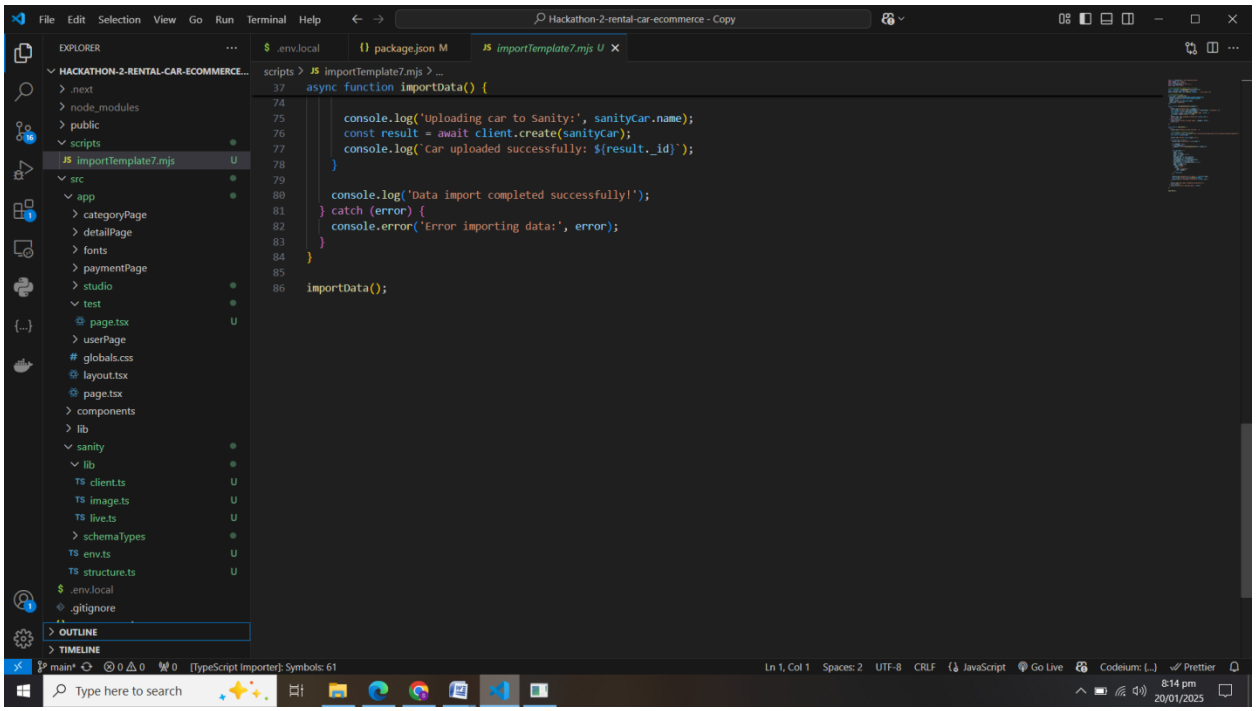
The exported JSON file was checked to make sure all the fields were filled correctly. This step ensured the data would be shown properly on the frontend when fetched.



```
import { createClient } from '@sanity/client';
import axios from 'axios';
import dotenv from 'dotenv';
import { fileURLToPath } from 'url';
import path from 'path';

// Load environment variables from .env.local
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
dotenv.config({ path: path.resolve(__dirname, '../.env.local') });

// Create Sanity client
const client = createClient({
  projectId: process.env.NEXT_PUBLIC_SANITY_PROJECT_ID,
  dataset: process.env.NEXT_PUBLIC_SANITY_DATASET,
  useCdn: false,
  token: process.env.SANITY_API_TOKEN,
  apiVersion: '2021-08-31'
});

async function uploadImageToSanity(imageUrl) {
  try {
    console.log(`Uploading image: ${imageUrl}`);
    const response = await axios.get(imageUrl, { responseType: 'arraybuffer' });
    const buffer = Buffer.from(response.data);
    const asset = await client.assets.upload('image', buffer, {
      filename: imageUrl.split('/').pop()
    });
    console.log(`Image uploaded successfully: ${asset._id}`);
    return asset._id;
  } catch (error) {
    console.error('Failed to upload image:', imageUrl, error);
    return null;
  }
}
```

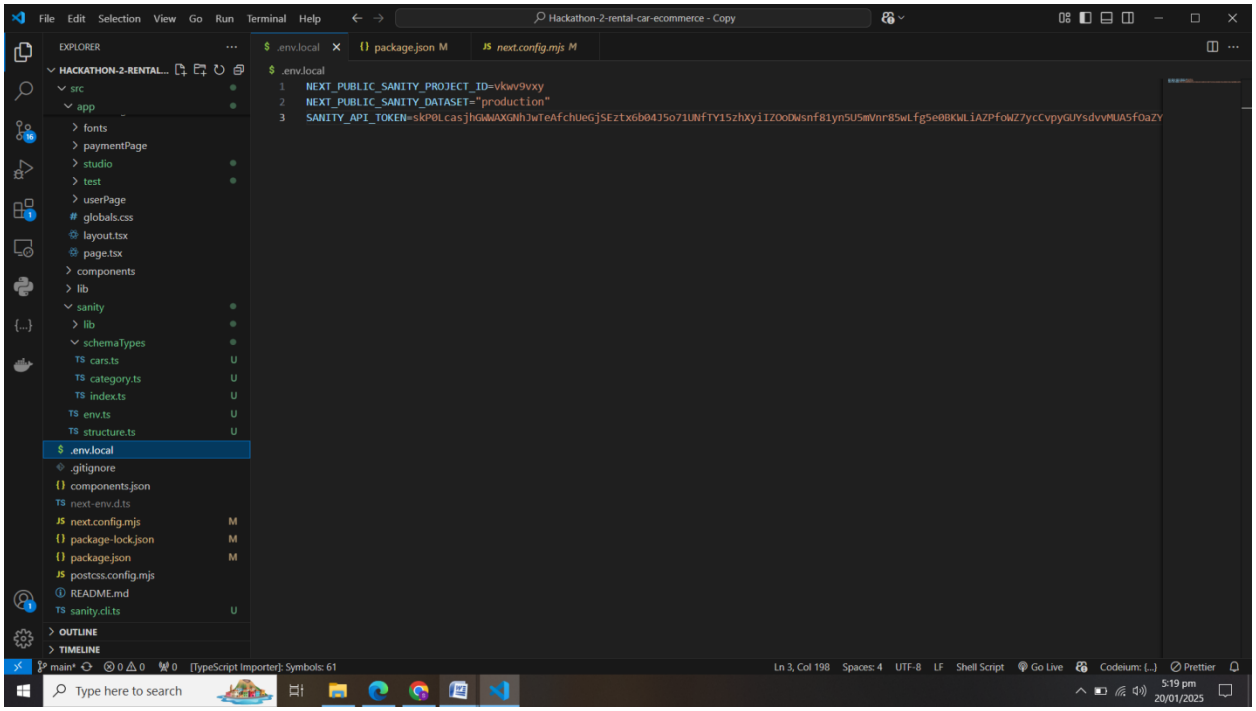


```
async function importData() {
  try {
    console.log('Fetching car data from API...');

    // API endpoint containing car data
    const response = await axios.get('https://sanity-nextjs-application.vercel.app/api/hackathon/template7');
    const cars = response.data;

    console.log(`Fetched ${cars.length} cars`);

    for (const car of cars) {
      console.log(`Processing car: ${car.name}`);

      let imageRef = null;
      if (car.image_url) {
        imageRef = await uploadImageToSanity(car.image_url);
      }

      const sanityCar = {
        _type: 'car',
        name: car.name,
        brand: car.brand || null,
        type: car.type,
        fuelCapacity: car.fuel_capacity,
        transmission: car.transmission,
        seatingCapacity: car.seating_capacity,
        pricePerDay: car.price_per_day,
        originalPrice: car.original_price || null,
        tags: car.tags || [],
        image: imageRef ? {
          _type: 'image',
          asset: {
            _type: 'reference',
            _ref: imageRef,
          },
        } : undefined,
```



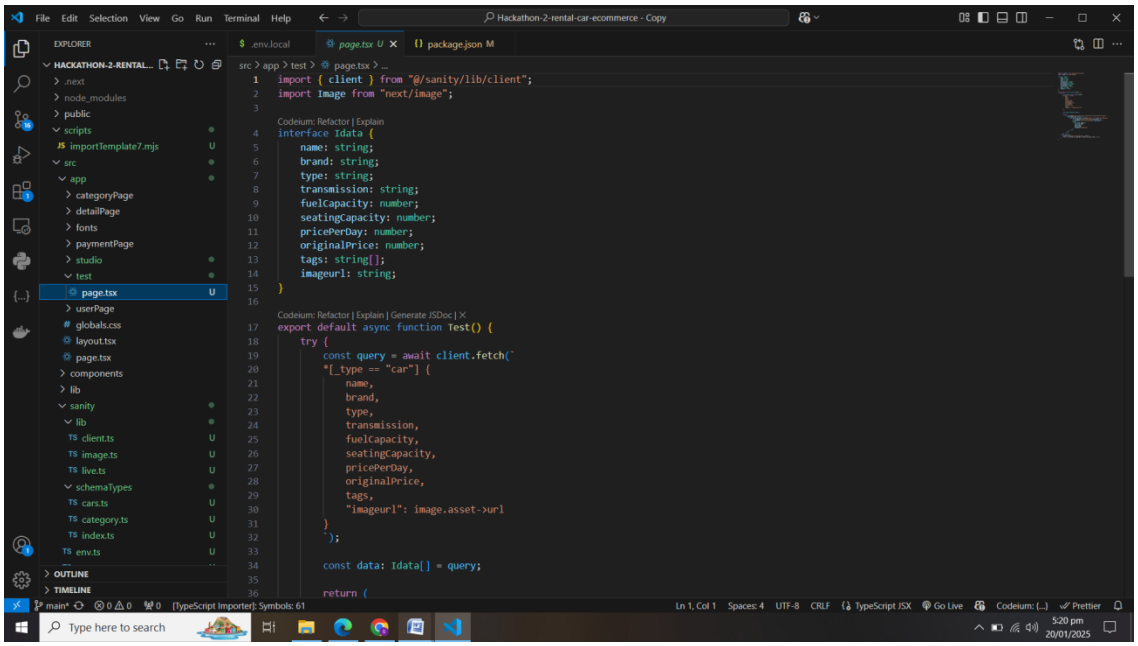
- **Environment Variables**

The .env file contains sensitive settings for the RentalHub application. Key entries:



- **Item Card Code**

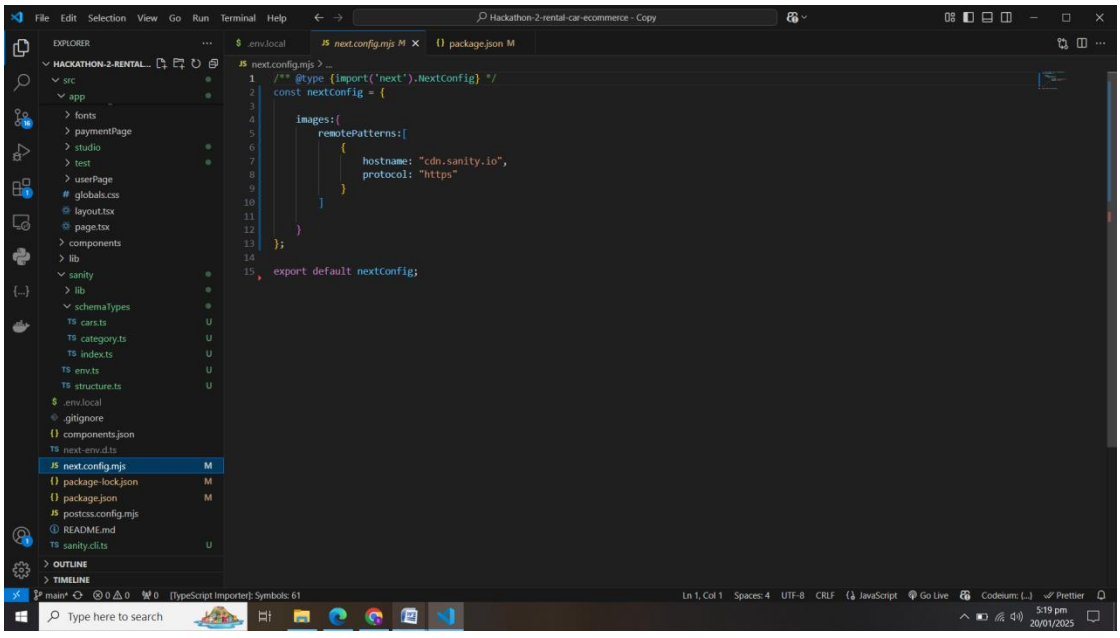
This code defines the design and functionality of a single product card. It is used on the client page to display individual items.



```
import { client } from "@/sanity/lib/client";
import Image from "next/image";

interface Idata {
    name: string;
    brand: string;
    type: string;
    transmission: string;
    fuelCapacity: number;
    seatingCapacity: number;
    pricePerDay: number;
    originalPrice: number;
    tags: string[];
    imageurl: string;
}

export default async function Test() {
    try {
        const query = await client.fetch(`
        *[_type == "car"] {
            name,
            brand,
            type,
            transmission,
            fuelCapacity,
            seatingCapacity,
            pricePerDay,
            originalPrice,
            tags,
            "imageurl": image.asset->url
        }
        `);

        const data: Idata[] = query;

        return (
```

- **Next.js Image Configuration**

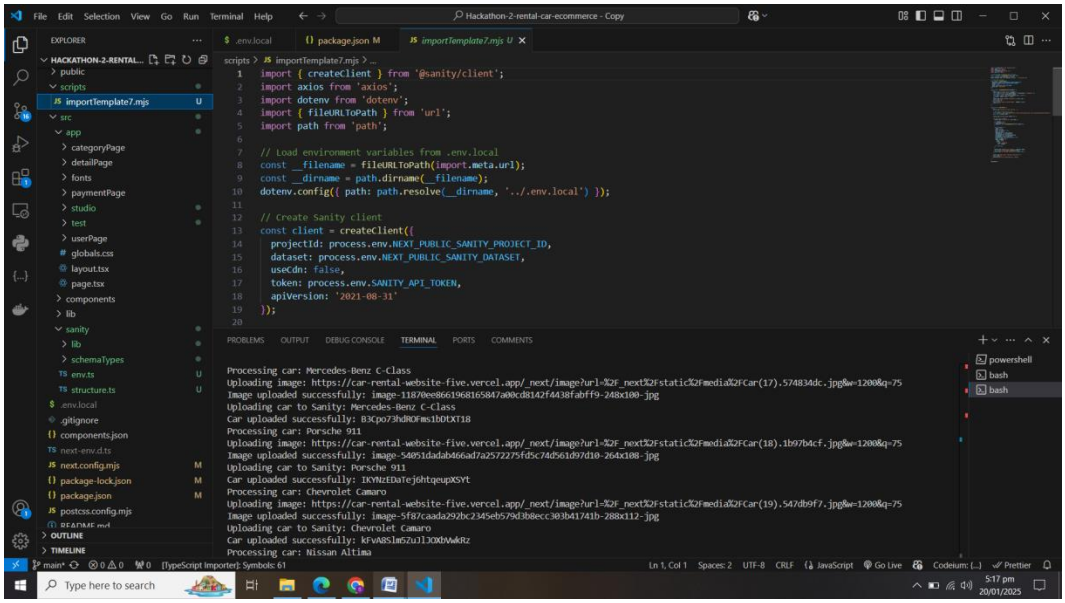
This code is used to allow images from an external source (like Sanity) to be used in a Next.js application. It specifies that images with the hostname "cdn.sanity.io" and the "https" protocol can be displayed in the app.



```
/** @type {import('next').NextConfig} */
const nextConfig = {

    images:{
        remotePatterns:[
            {
                hostname: "cdn.sanity.io",
                protocol: "https"
            }
        ]
    }
};

export default nextConfig;
```

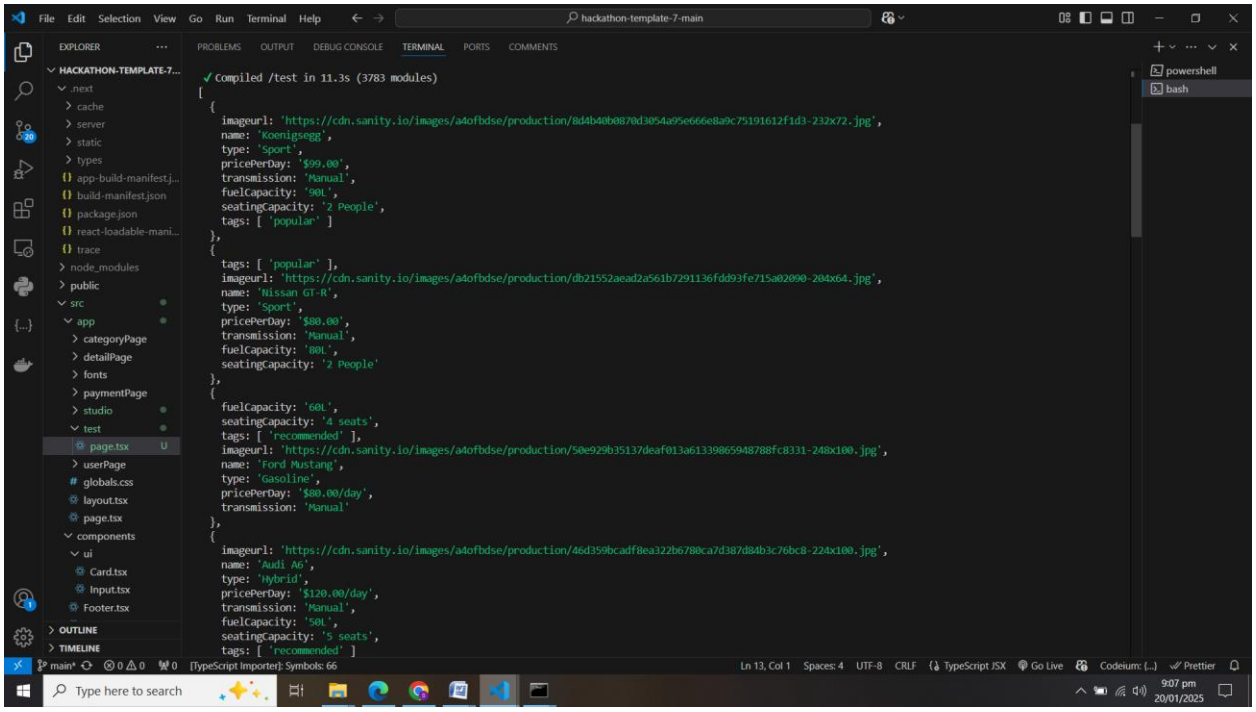
- Successfully Migrated Data Into Sanity:

The data was successfully moved into Sanity, ensuring all product information was correctly placed and ready to use.



- **Re-importing Data:**

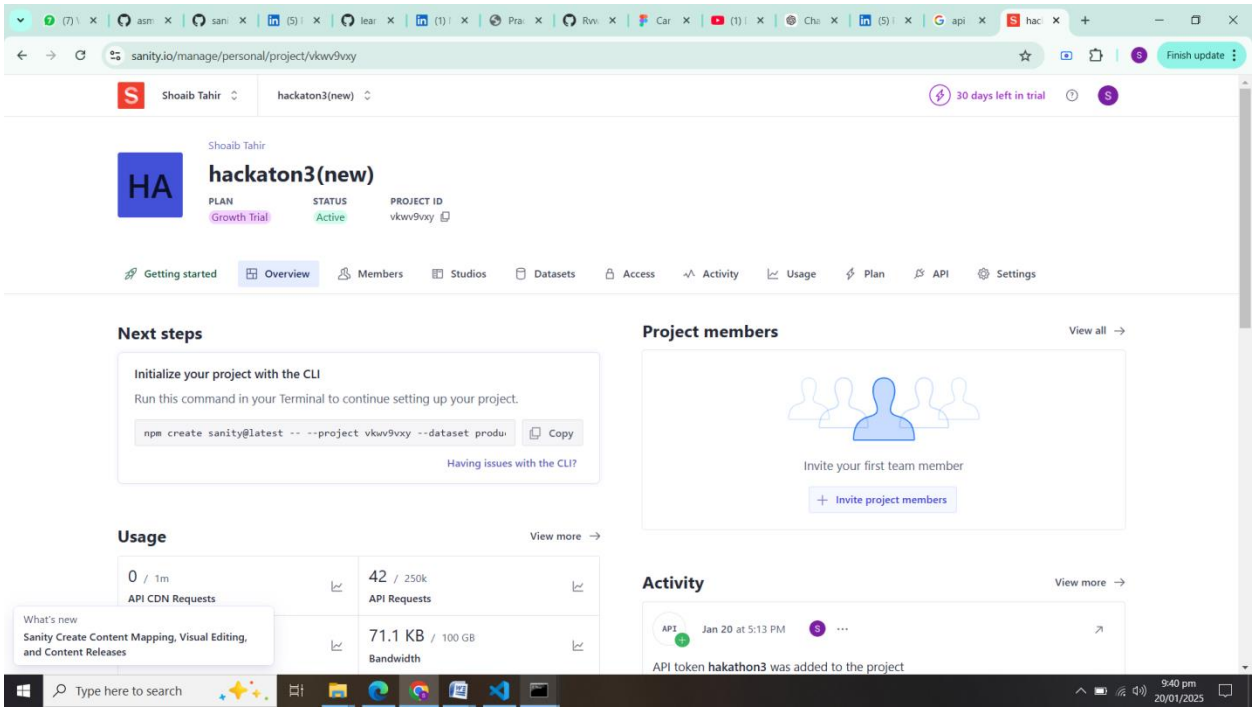
After checking the data, it was re-imported into Sanity CMS. This confirmed that the data migration was successful and everything was working as expected.



4. Tools Used:

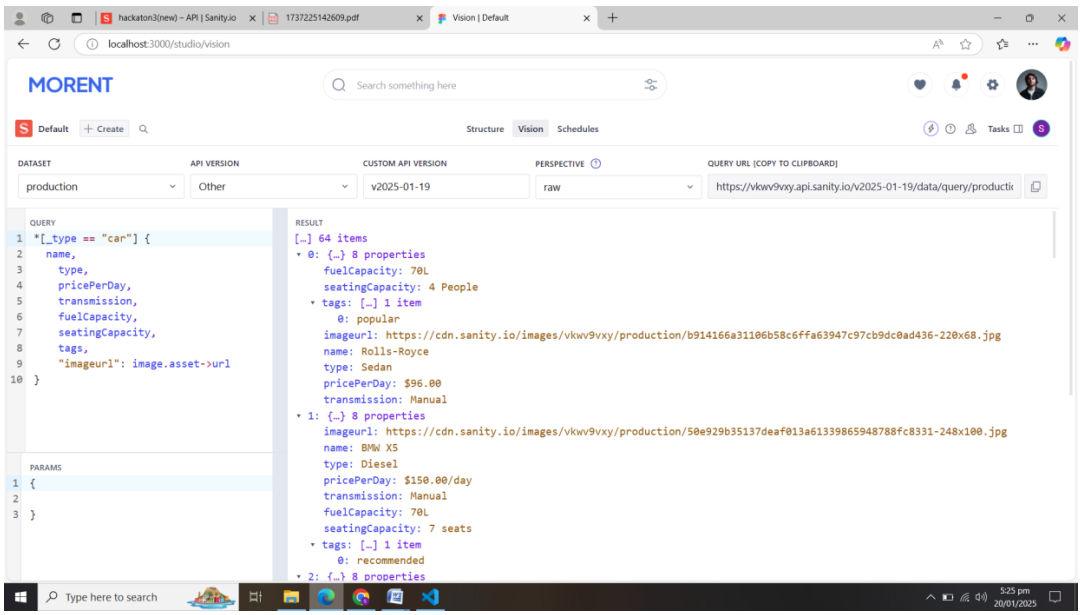
- **Sanity Studio:**

Used to create schemas, manage content, and display car data.



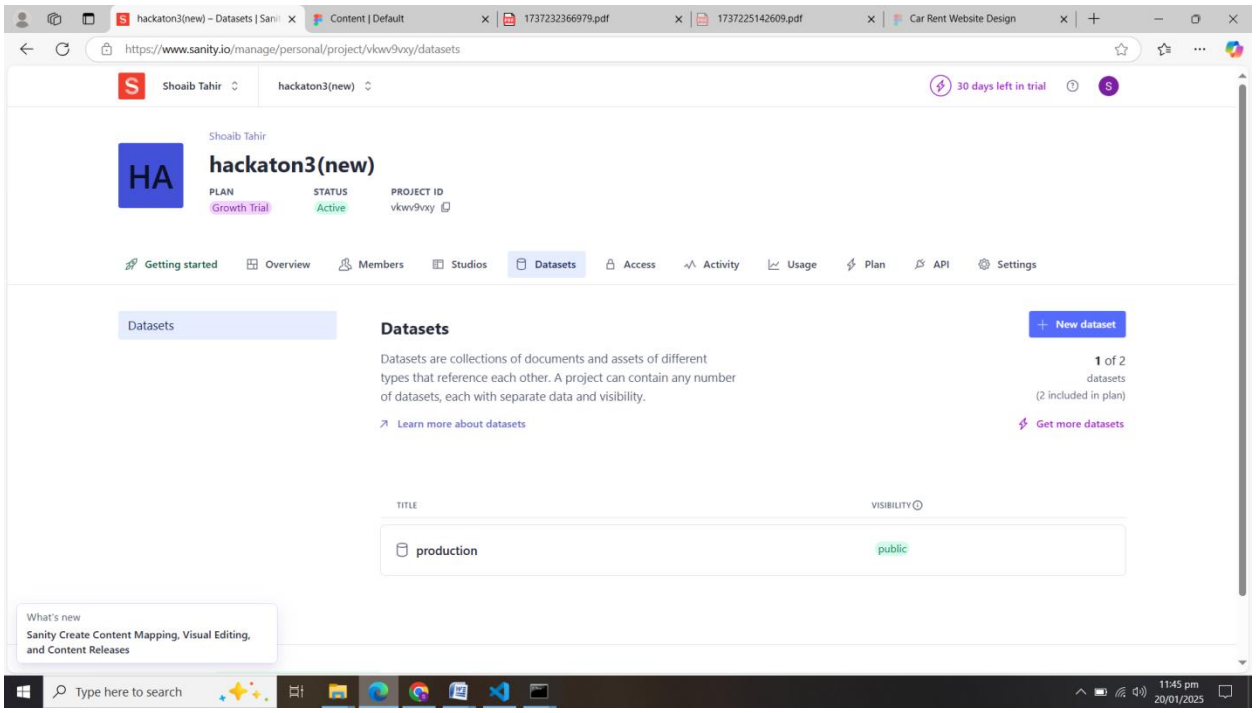
- **Sanity Vision:**

A tool used for testing and debugging GROQ queries to fetch and preview data in Sanity Studio.



- **Sanity Database:**

A cloud-based database used to store and manage structured content, like product data, for the application.



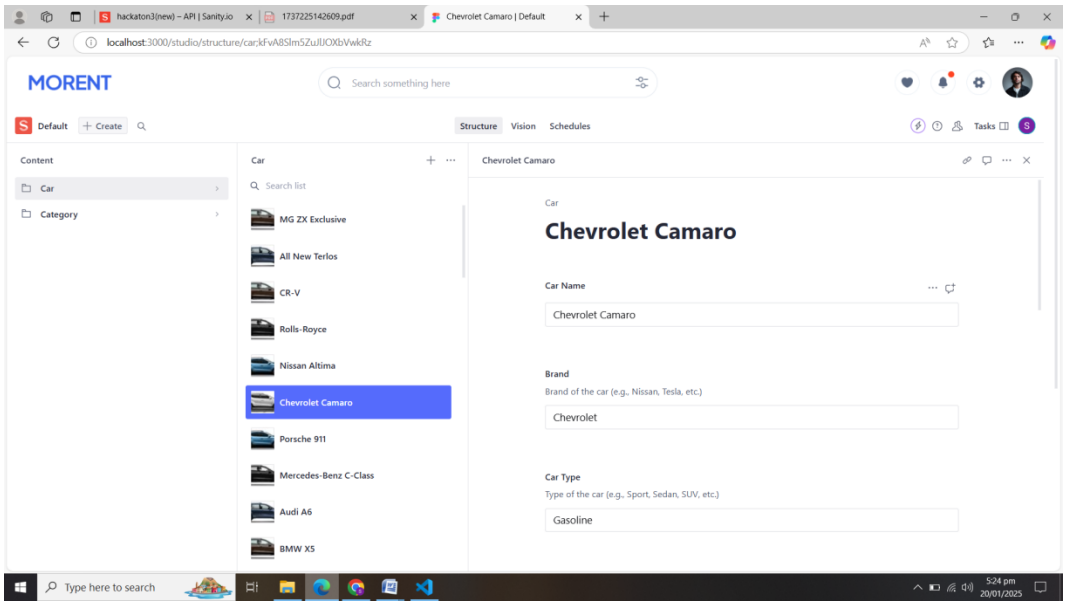
- **Sanity CLI:**

Used for exporting and importing datasets to ensure data is consistent and backed up.

5. Screenshots and Frontend Display:

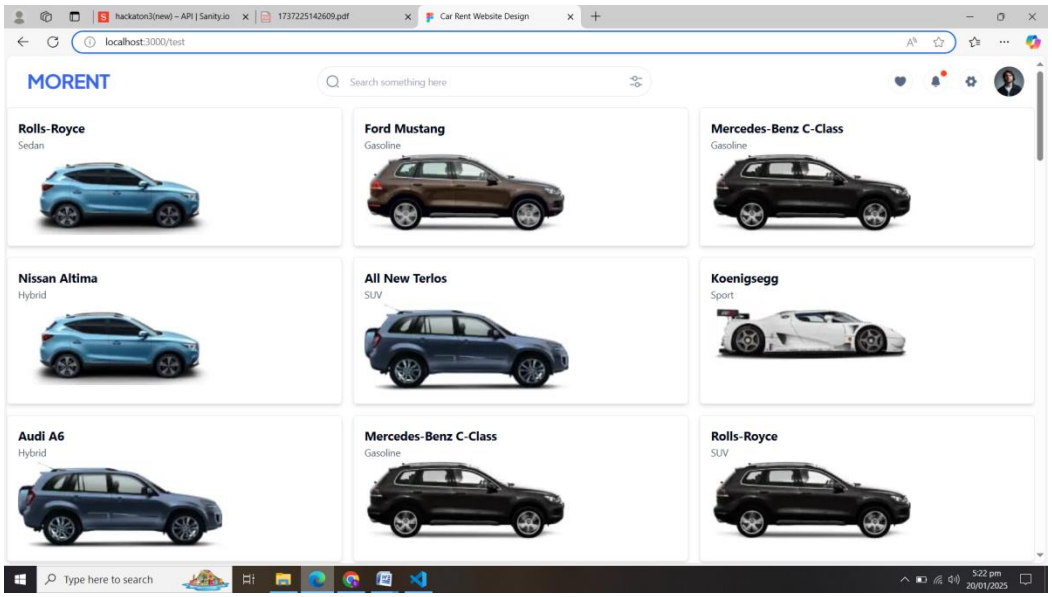
- **Sanity CMS Fields**:

A screenshot showing the filled fields in Sanity Studio, displaying car details like name, brand, type, fuel capacity, price per day, and images.



- **Frontend Display:**

A screenshot showing how the car data is dynamically displayed on the frontend of the rental marketplace.



Key Steps for Sanity API Integration and Data Migration

1. Sanity API Setup:

- Connects to Sanity's API using a dataset and project ID with an API token for security.
- Environment variables (like project ID and dataset) keep sensitive information safe.

2. Fetching Data from Sanity:

- Uses GROQ queries to get structured data from Sanity CMS.
- Example: Queries fetch car categories, descriptions, prices, and other details.

3. Mapping and Formatting:

- Matches the fetched data with the RentalHub schema.
- Formats records to fit the application’s requirements.

4. Saving to Database:

- Saves data into the RentalHub database through REST API calls or direct commands.
- Handles errors to log issues without stopping the process.

5. Code Highlights:

- Reusability: Functions can be reused for future migrations.
- Efficiency: Bulk data insertion reduces API calls and works faster.

6. Conclusion:

The API integration and data migration were successfully completed, improving the efficiency and scalability of the RentalHub project. This integration made it easier to add and update car data in the marketplace, while the migration steps ensured data accuracy and consistency throughout the system. With this setup, the RentalHub project is now more dynamic and easier to manage.