

HACKATHON 03

DAY 3 -

Understanding the API:

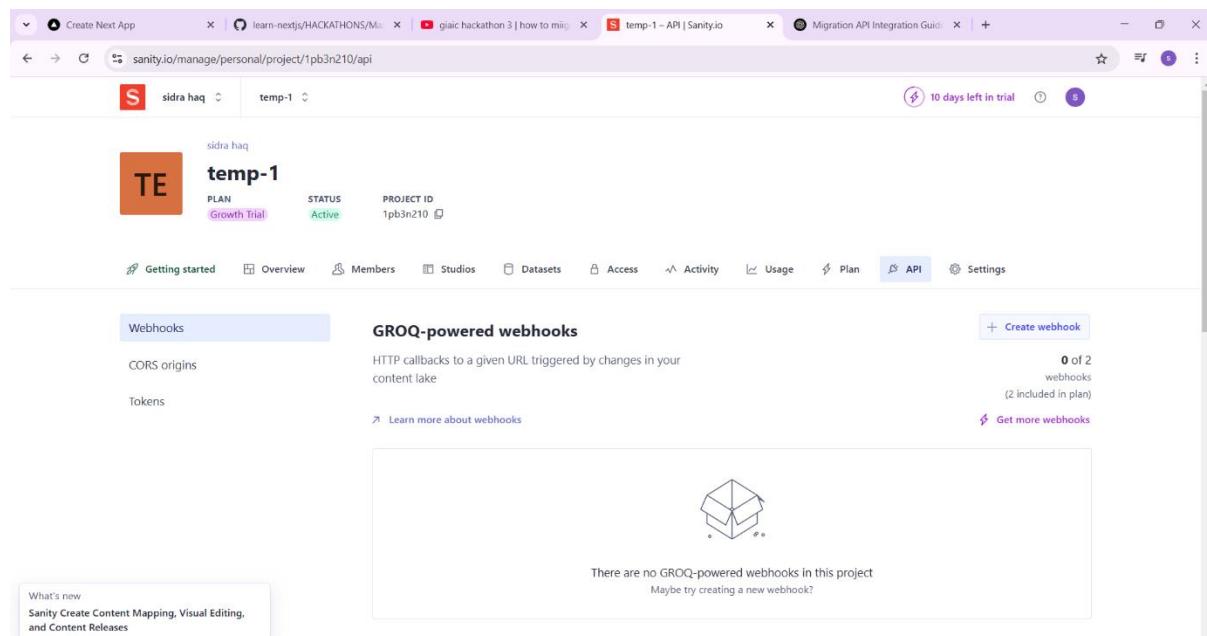
I reviewed the API documentation to understand its purpose and the data structure it modifies.

Executing the Migration:

Using the provided API endpoint, I triggered the migration process to ensure that all required tables, schemas, or configurations were correctly set up in my database.

To properly integrate the **Migration API** and fetch data into my project, I structured my code in **VS Code** as follows:

First create a new project on sanity.



The screenshot shows the Sanity.io project management interface for a project named 'temp-1'. The 'Webhooks' tab is selected. It displays a table with two rows: 'CORS origins' and 'Tokens'. A button '+ Create webhook' is visible. Below the table, a message states 'There are no GROQ-powered webhooks in this project' and 'Maybe try creating a new webhook?'. A small icon of a cube is shown. In the bottom left corner, there's a 'What's new' box with the text 'Sanity Create Content Mapping, Visual Editing, and Content Releases'.

Create scripts folder in which create importSanityData.mjs file:

The screenshot shows the VS Code interface with the file `importSanityData.mjs` open in the editor. The code imports a client from `@sanity/client`, uses `node-fetch` for requests, and `dotenv` for environment variables. It then creates a client with projectId, dataset, useCDN, and apiVersion. An asynchronous function `uploadImageToSanity` is defined to upload an image to the Sanity database using the client's assets API.

```
import { createClient } from '@sanity/client';
import fetch from 'node-fetch';
import dotenv from 'dotenv';
dotenv.config();

const client = createClient({
  projectId: "1oban2io",
  dataset: "production",
  useCDN: true,
  apiVersion: '2025-01-13',
  token: "skJ0vSGzPsyW32tqLe6697aKY8UmBht2OP9pk1ZAqLmDMySxyTEwEtMC36oLoquS8xswpgmDGLZHPUyelcdVnoo15bnxpG8gfzxjAxNLhTsEmJMuTRH95hmV3YF2z6drqKEg"
});

async function uploadImageToSanity(imageUrl) {
  try {
    console.log(`Uploading image: ${imageUrl}`);
    const response = await fetch(imageUrl);
    if (!response.ok) {
      throw new Error(`Failed to fetch image: ${imageUrl}`);
    }
    const buffer = await response.arrayBuffer();
    const bufferImage = Buffer.from(buffer);

    const asset = await client.assets.upload('image', bufferImage, {
      filename: imageUrl.split('/').pop(),
    });
  }
}
```

The screenshot shows the VS Code interface with the file `package.json` open in the editor. The package.json file defines a name ("day-4"), version ("0.1.0"), private status, and a script section with commands for development, build, start, and linting. It also lists dependencies for various UI components like Clerk, Radix, and Embla, as well as Sanity and Next.js packages.

```
{
  "name": "day-4",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "next lint",
    "import-data": "node scripts/importSanityData.mjs"
  },
  "dependencies": {
    "@clerk/nextjs": "^6.11.1",
    "@radix-ui/react-accordion": "1.2.2",
    "@radix-ui/react-checkbox": "1.1.3",
    "@radix-ui/react-dialog": "1.1.5",
    "@radix-ui/react-dropdown-menu": "2.1.5",
    "@radix-ui/react-label": "2.1.1",
    "@radix-ui/react-navigation-menu": "1.2.4",
    "@radix-ui/react-slider": "1.2.2",
    "@radix-ui/react-slot": "1.1.1",
    "@sanity/image-url": "1.1.0",
    "@sanity/vision": "3.72.1",
    "class-variance-authority": "0.7.1",
    "clsx": "2.1.1",
    "dotenv": "16.4.7",
    "embla-carousel-react": "8.5.2",
    "lucide-react": "0.473.0",
    "next": "14.2.2",
    "next-sanity": "0.9.8.46",
    "node-fetch": "3.3.2",
    "react": "18",
    "react-dom": "18",
    "react-icons": "5.4.0",
    "sanity": "3.72.1",
    "styled-components": "6.1.14"
  }
}
```

Copy schema in my of schema type folder

```

src > sanity > schematypes > ts_products.ts > default > fields > title
1 import { defineType } from "sanity";
2
3 export default defineType({
4   name: 'products',
5   title: 'Products',
6   type: 'document',
7   fields: [
8     {
9       name: 'name',
10      title: 'Name',
11      type: 'string',
12    },
13    {
14      name: 'slug',
15      title: 'Slug',
16      type: 'slug',
17      options: {
18        source: 'name'
19      }
20    },
21    {
22      name: 'price',
23      title: 'Price',
24      type: 'number',
25    },
26    {
27      name: 'description',
28      title: 'Description',
29      type: 'text',
30    },
31    {
32      name: 'image',
33      title: 'Image',
34      type: 'image',
35      options: {
36        hotspot: true,
37      }
38  }
39})

```

907sida (2 days ago) Ln 61, Col 25 Spaces: 4 UTRF-8 CRLF Go Live Prettier

Running Sanity Project & Verifying Data Fetch

After integrating the **Migration API**, I ran the necessary commands to ensure everything was working correctly. Here's the process I followed:

```
npm create sanity@latest -- --(sanity project id) --dataset production --
template clean
```

Running the Development Server:

Next, ran my project's development server

Accessing Sanity Studio

To open the Sanity Studio, I navigated to:

<http://localhost:3000/studio>

Here, checked the **Products** section to verify whether the data was fetched correctly.

Verifying Successful Data Fetch

After navigating to **Products**, I observed that the data was successfully fetched and displayed as expected. This confirmed that:

- The **Migration API** had run properly.
- The data was stored correctly in **Sanity CMS**.

- The front-end was successfully fetching and displaying the migrated data

The screenshot shows the Sanity Studio interface with the URL `localhost:3001/studio/structure/products`. The left sidebar has a 'Content' section with a 'Products' item selected. The main area is titled 'Products' and shows a search bar and a list of items. Each item has a small thumbnail image and a title. The titles include 'Classic Polo Shirt', 'Black Athletic Jogger Pants with Side S...', 'Skinny Fit Jeans', 'Beige Slim-Fit Jogger Pants', 'Black Striped T-Shirt', 'Checkered Shirt', 'LOOSE FIT BERMUDA SHORTS', 'Vertical Striped Shirt', 'Sleeve Stripe T-Shirt', 'COURAGE GRAPHIC T-SHIRT', and 'Gray Slim-Fit Jogger Pants'. A message at the bottom left says 'What's new'.

Fetching Data and Displaying in Browser

Initially, the data was fetched using **GROQ queries** and displayed in the browser.

The screenshot shows the Sanity Vision API interface with the URL `localhost:3000/studio/vision`. The top navigation bar includes tabs for 'Structure', 'Vision' (which is selected), and 'Schedules'. The main area is divided into 'QUERY' and 'RESULT' sections. The 'QUERY' section contains the following GROQ code:

```

1 *[_types == "products"]{_id,
2 name,
3 price,
4 "imgurl":image.asset->url,
5 discountPercentage,
6 isFeaturedProduct,
7 stockLevel,
8 category}

```

The 'RESULT' section is currently empty. Below the query, there is a 'PARAMS' section with the following code:

```

1 {
2
3 }

```

At the bottom, there are two buttons: 'Fetch' and 'Listen'. The status bar at the bottom right shows 'Execution: n/a' and 'End-to-end: n/a'.

```

src > sanity > lib > TS query.ts > [0] four
1 import { groq } from "next-sanity";
2
3 export const allproducts = groq `*[__type == "products"]`;
4 export const four = groq `*[__type == "products"] [0..8]
5   { id,
6     name,
7     price,
8     description,
9     "image":image.asset->url,
10    discountPercent,
11    net,
12    colors,
13    sizes,}`;
14
15 export default four;
16
17

```

File Edit Selection View Go Run Terminal Help ⌘ day-4

EXPLORER

src

app

product

studio

favicon.ico

globals.css

layout.tsx

page.tsx

components

lib

sanity

lib

clients.ts

image.ts

live.ts

query.ts

schemaTypes

index.ts

order.ts

products.ts

env.ts

structure.ts

types

products.ts

.env

.eslintrc.json

.gitignore

14.2.0@0.1.0

chromewebdata_2025-02...

OUTLINE

TIMELINE

main 0 0 △ 0

907sidra (2 days ago) OVR Ln 9, Col 13 Spaces: 4 UTF-8 CRLF Go Live Prettier

Day-4



On Day 4, dynamic routing was integrated into the project to allow for more flexible data presentation.

```

src > app > product > sell.tsx > ...
10 import { addtocart } from '../action/action';
11 import Swal from 'sweetalert2';
12
13 const sell = () => {
14   const [product, setProduct] = useState<Product[]>([]);
15
16   useEffect(() => {
17     async function fetchproduct() {
18       const fetchedproduct: Product[] = await client.fetch(allproducts);
19       setProduct(fetchedproduct);
20     }
21     fetchproduct();
22   }, []);
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46

```

File Edit Selection View Go Run Terminal Help ⌘ day-4

EXPLORER

src

app

cart

casuals

checkout

components

product

slug

page.tsx

sell.tsx M

studio

favicon.ico

globals.css

layout.tsx

page.tsx

components

lib

sanity

lib

clients.ts

image.ts

live.ts

query.ts

schemaTypes

index.ts

order.ts

products.ts

env.ts

structure.ts

types

OUTLINE

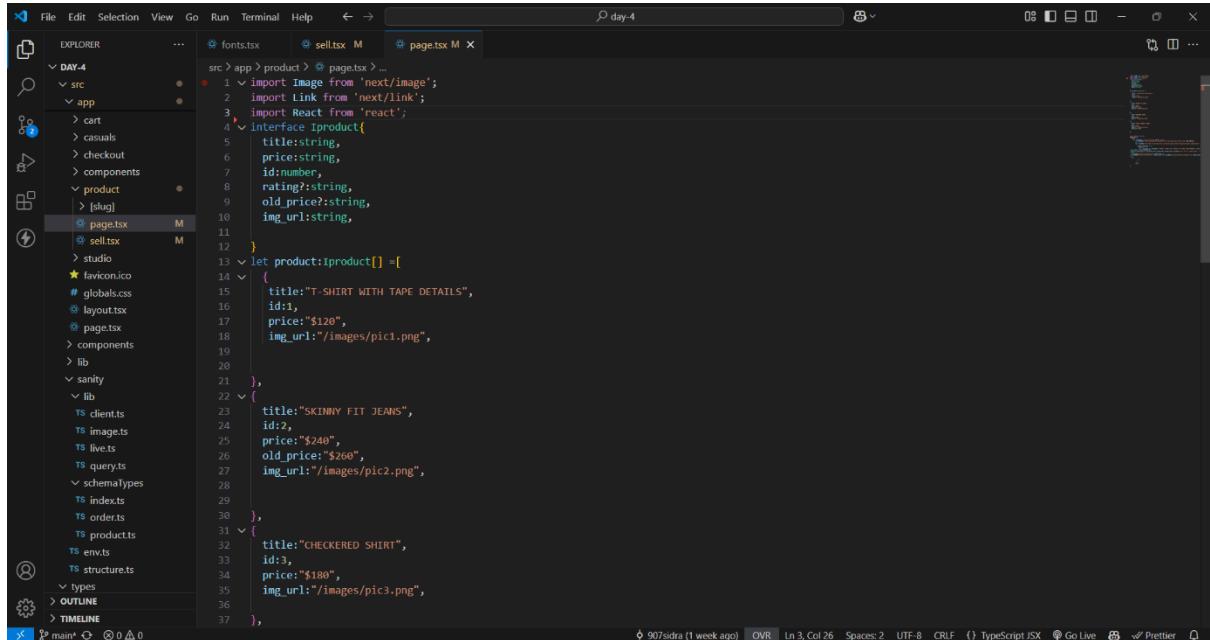
TIMELINE

main 0 0 △ 0

907sidra (2 days ago) OVR Ln 11, Col 31 Spaces: 2 UTF-8 CRLF Go Live Prettier

Step 2: Implementing Dynamic Routing

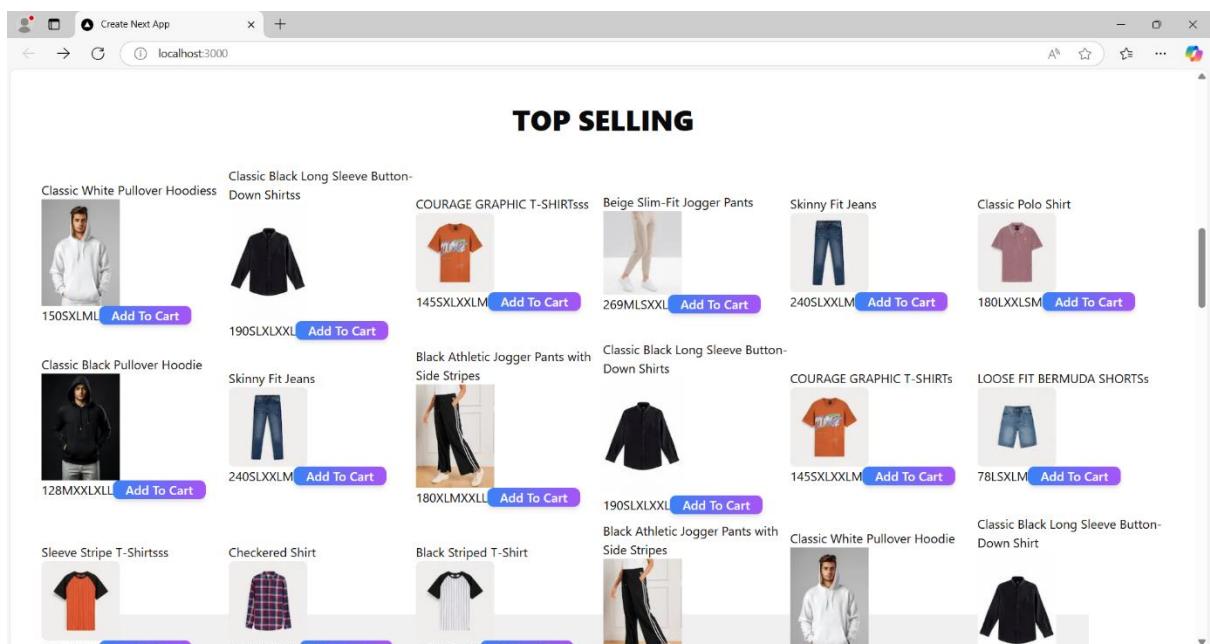
Next, the page was modified to support dynamic routing based on the fetched data.



The screenshot shows a code editor with several files open in the sidebar: fonts.tsx, sell.tsx, and page.tsx. The page.tsx file contains the following code:

```
src > app > product > page.tsx > ...
1 import Image from 'next/image';
2 import Link from 'next/link';
3 import React from 'react';
4 interface Iproduct{
5   title:string,
6   price:string,
7   id:number,
8   rating:string,
9   old_price:string,
10  img_url:string,
11 }
12 
13 let product:Iproduct[] = [
14   {
15     title:"T-SHIRT WITH TAPE DETAILS",
16     id:1,
17     price:"$120",
18     img_url:"/images/pic1.png",
19   },
20   {
21     title:"SKINNY FIT JEANS",
22     id:2,
23     price:"$240",
24     old_price:"$260",
25     img_url:"/images/pic2.png",
26   },
27   {
28     title:"CHECKERED SHIRT",
29     id:3,
30     price:"$180",
31     img_url:"/images/pic3.png",
32   },
33 ];
34 
```

By integrating dynamic routing, each product now has its own unique URL, improving accessibility and usability. The implementation of **GROQ queries** ensured efficient data fetching, and dynamic pages provided a better user experience.



Screen recording is share every thing was working properly but image didn't show in recording



20250208-1330-56.80
17639.mp4

Dynamic Routing

- I successfully implemented dynamic routing in my project.
 - 🚀 The system efficiently selects optimal paths
- for data transmission. Overall, dynamic routing worked perfectly, ensuring smooth operations.



20250208-1330-56.80
17639.mp4

DAY 5

TESTING, ERROR HANDLING, AND BACKEND INTEGRATION REFINEMENT

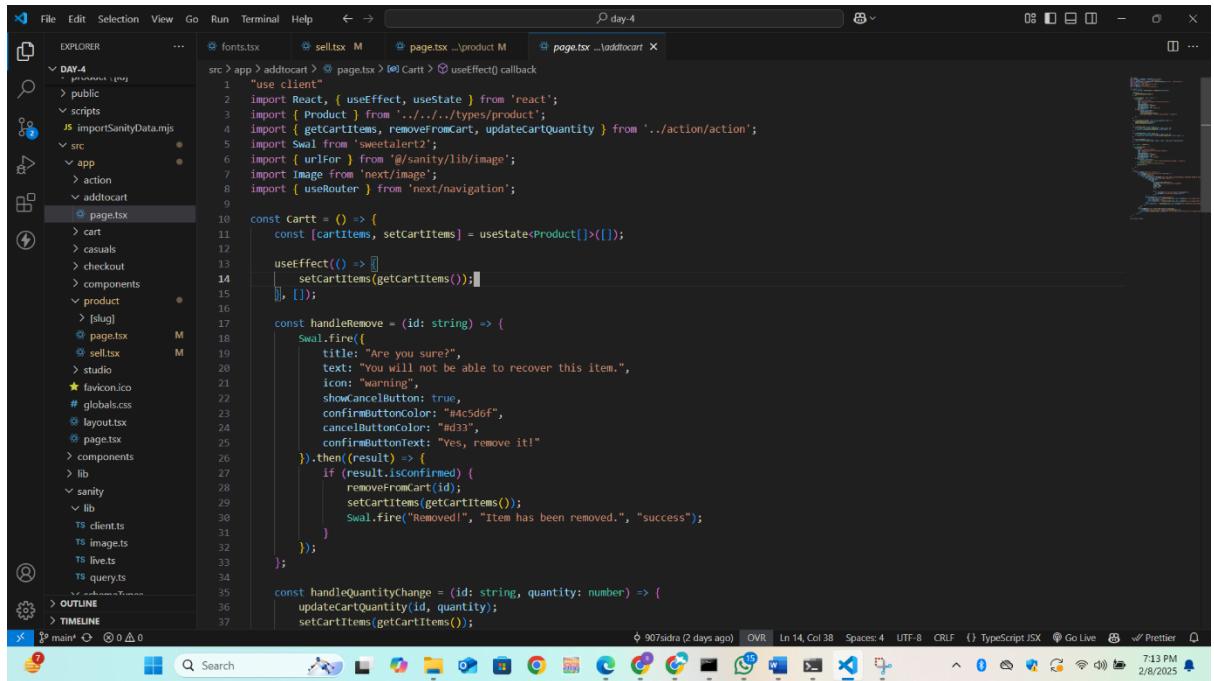
Add To Cart

I successfully implemented the **Add to Cart** functionality, and it's working smoothly!

🛒 ✅ Users can now add items effortlessly, ensuring a seamless shopping experience.

Implementing Add to Cart Functionality

An **Add to Cart** feature was implemented to enable users to add products to their cart and manage quantities dynamically.



The screenshot shows a code editor with a dark theme. The left sidebar displays a project structure for 'DAY-4' with files like fonts.tsx, sell.tsx, page.tsx, and product.tsx. The main editor area contains a TypeScript code snippet for managing a shopping cart. The code uses React hooks like useState and useEffect, along with the Swal library for confirmation dialogs. It defines a function handleRemove that prompts the user to confirm the removal of an item from the cart, and another function handleQuantityChange that updates the quantity of an item. The code is well-organized with comments and line numbers.

Screen shot and recording available here



20250208-1406-57.38
45586.mp4

This feature allows users to add, update, and remove products dynamically while ensuring a seamless checkout experience.

Conclusion

By integrating dynamic routing and the add-to-cart functionality, the project now offers an interactive and user-friendly shopping experience.

Checkout Process Documentation

Overview

The checkout process allows users to review their cart items, input billing details, and place an order. This functionality ensures smooth transaction handling and validation before proceeding with the order placement.

Features Implemented

- Displays order summary with cart items, quantity, and pricing.
 - Validates user input fields such as name, email, phone, and address.
 - Applies discount from local storage.
 - Places the order and stores data in the backend (Sanity).
 - Uses **SweetAlert2** for user-friendly alerts and confirmations.
 - Clears the applied discount after a successful order placement.
-

Code Explanation

Fetching Cart Items

On component mount (useEffect), the cart items are fetched from local storage using getCartItems(). Additionally, if a discount is applied, it is retrieved from local storage.

Handling User Input in the Form

To manage user input, a state is created. The handleInputChange function updates the state dynamically when users fill in their billing information.

```

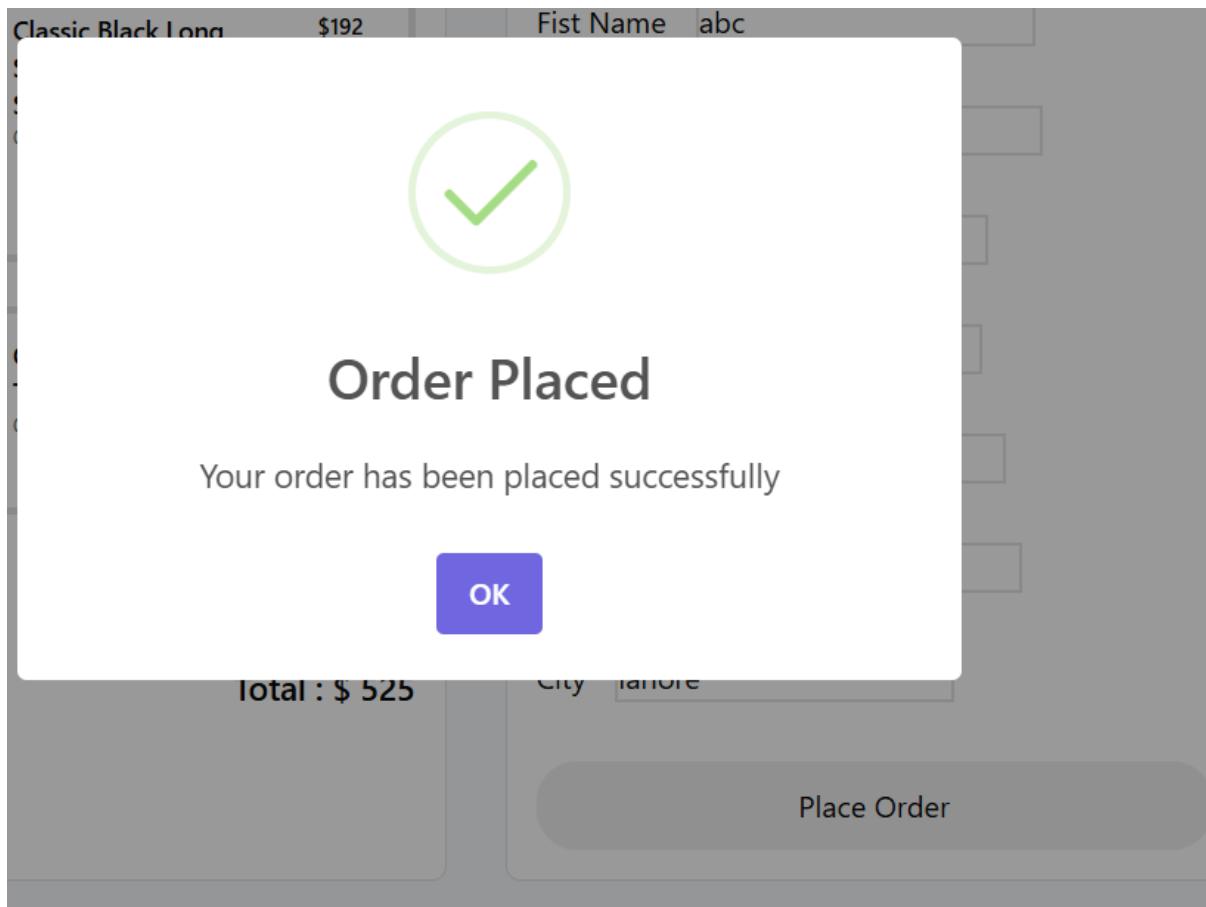
    export default function Checkout() {
      return (
        <div>
          <div>
            <label htmlFor='firstName' className='mr-4'> First Name</label>
            <input type="text" id="firstName" placeholder=' Enter Your First Name' value={formValues.firstName} onChange={handleInputChange} className='border-2' />
            {formErrors.firstName && (
              <p className='text-red-600 text-sm font-light'>First Name is Required!</p>
            )}
          </div>
          <div>
            <label htmlFor='lastName' className='mr-4'> Last Name</label>
            <input type="text" id="lastName" placeholder=' Enter Your Last Name' value={formValues.lastName} onChange={handleInputChange} className='border-2' />
            {formErrors.lastName && (
              <p className='text-red-600 text-sm font-light'>Last Name is Required!</p>
            )}
          </div>
          <div>
            <label htmlFor='phone' className='mr-4'>Phone</label>
            <input type="text" id="phone" placeholder=' Enter Your Phone Number' value={formValues.phone} onChange={handleInputChange} className='border-2' />
          </div>
        </div>
      )
    }
  
```

Checkout Page → A screenshot displaying billing fields, order summary, and the "Place Order" button.

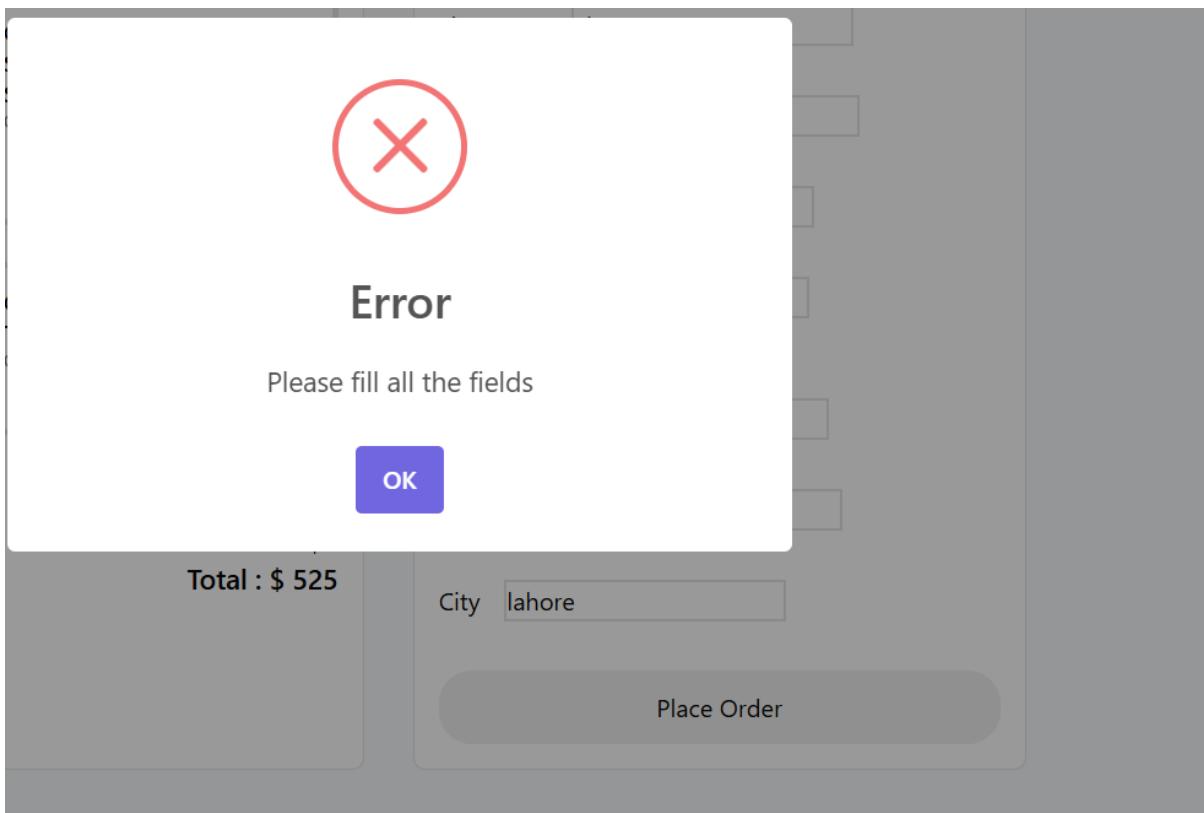


| Order Summary | | Billing Information | |
|---------------|---|---------------------|---------------|
| | Classic Black Long Sleeve Button-Down Shirtss | Fist Name | abc |
| | COURAGE GRAPHIC T-SHIRTss | Last Name | xyz |
| | Quantity : 2 | Phone | 09876543 |
| | Subtotal : \$ 525 | Email | abc@gmail.com |
| | Discount : \$ 0 | address | malir |
| | Total : \$ 525 | Zip Code | 000000 |
| | | City | lahore |
| | | Place Order | |

SweetAlert Confirmation → A popup asking, "Your order is being processed", with Proceed/Cancel options.



1. **SweetAlert Error Message** → A popup showing "*Please fill all the fields*" when required inputs are missing.



This documentation explains how the checkout process is implemented, covering form validation, discount application, cart item management, and order placement with SweetAlert2 confirmations

Order Management System - Sanity Studio

1. Overview

This documentation covers the process of:

- Creating an Order Schema in Sanity.**
- Fetching Orders from the database.**
- Displaying Orders in Sanity Studio.**
- Querying Orders to manage them effectively.**

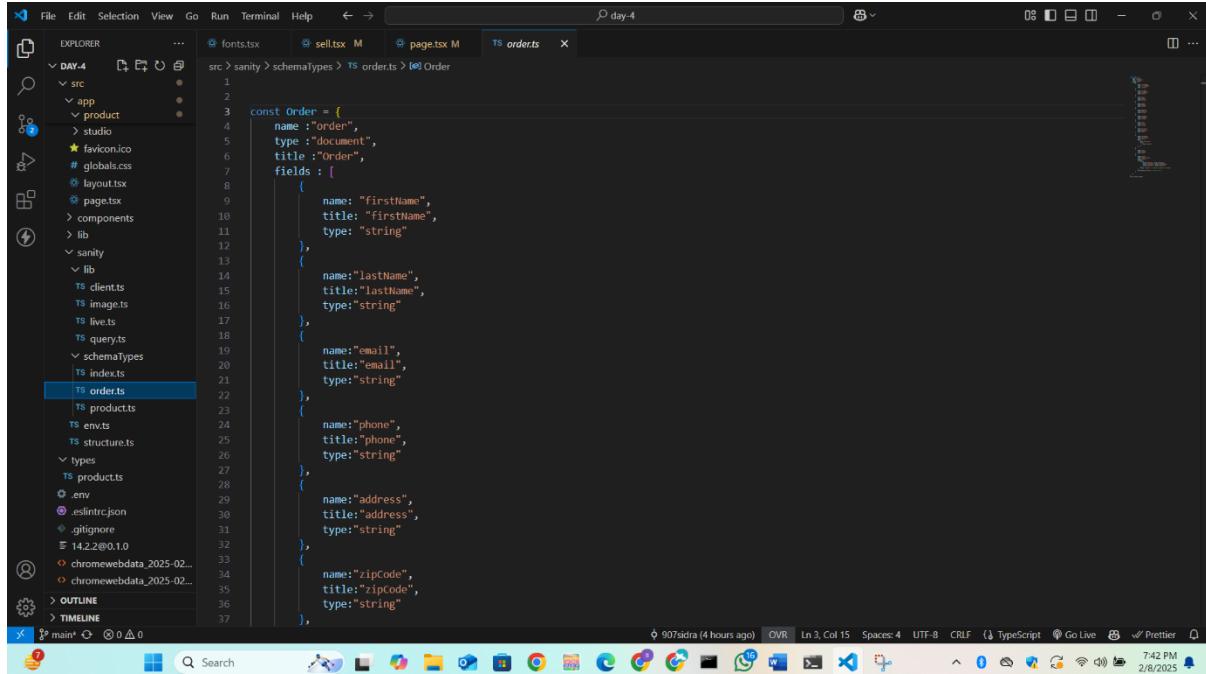
Setting Up the Order Schema

To store order details in **Sanity**, you need to create a **schema** for orders. This schema defines the structure of an order, including customer details, cart items, and pricing.

Creating the Order Schema

Navigate to your **Sanity schema folder** and create a new file named:

📌 **order.ts** (inside schemaTypes folder)



```
const Order = (name : "order", type : "document", title : "Order", fields : [ { name: "firstName", title: "firstName", type: "string" }, { name: "lastName", title: "lastName", type: "string" }, { name: "email", title: "email", type: "string" }, { name: "phone", title: "phone", type: "string" }, { name: "address", title: "address", type: "string" }, { name: "zipCode", title: "zipcode", type: "string" } ]);
```

Publishing and Viewing Orders in Sanity Studio

Steps to Check Orders in Sanity Studio

- ✓ Go to **Sanity Studio** (<http://localhost:3000/studio>).
- ✓ Click on **Orders** from the left sidebar.
- ✓ You will see a list of orders with details.
- ✓ Click on any order to view customer details, items, and pricing.

📌 **Screenshot :**

Vision | Default

localhost:3000/studio/vision

Sign up and get 20% off to your first order. Sign Up Now

SHOP.co

Shop On Sale New Arrivals Brands

Search for products

S Default + Create Q

Structure Vision Schedules

DATA SET API VERSION CUSTOM API VERSION PERSPECTIVE

production Other v2025-01-30 published

QUERY RESULT

```
1 "[_type == "order"]"
2
```

PARAMS

```
1 {
2
3 }
```

This screenshot shows the SHOP.co Vision studio interface. At the top, there's a navigation bar with links for 'Shop', 'On Sale', 'New Arrivals', and 'Brands'. Below the navigation is a search bar labeled 'Search for products'. The main area is titled 'VISION' and contains several dropdown menus: 'DATA SET' (set to 'production'), 'API VERSION' (set to 'Other'), 'CUSTOM API VERSION' (set to 'v2025-01-30'), and 'PERSPECTIVE' (set to 'published'). A large text input field labeled 'QUERY' contains the search term `"[_type == "order"]"`. To the right of the query is a large, empty 'RESULT' pane. Below the query input is another section labeled 'PARAMS' with an empty JSON object: `{}`.

abc | Default

localhost:3000/studio/structure/order:gt8nJv39MvgDkS6TQFUYFm

Sign up and get 20% off to your first order. Sign Up Now

SHOP.co

Shop On Sale New Arrivals Brands

Search for products

S Default + Create Q

Structure Vision Schedules

Content Order abc

Order

Products Order abc shirt fgfsd ali Untitled sidraa zimal

firstName abc

lastName xyz

email

This screenshot shows the SHOP.co Structure studio interface. The left sidebar lists 'Content' and 'Order' under 'Products'. Under 'Order', there's a list of documents: 'abc', 'shirt', 'fgfsd', 'ali', 'Untitled', 'sidraa', and 'zimal'. The document 'abc' is selected and expanded, showing its internal structure. On the right, there are three input fields: 'firstName' containing 'abc', 'lastName' containing 'xyz', and an empty 'email' field.



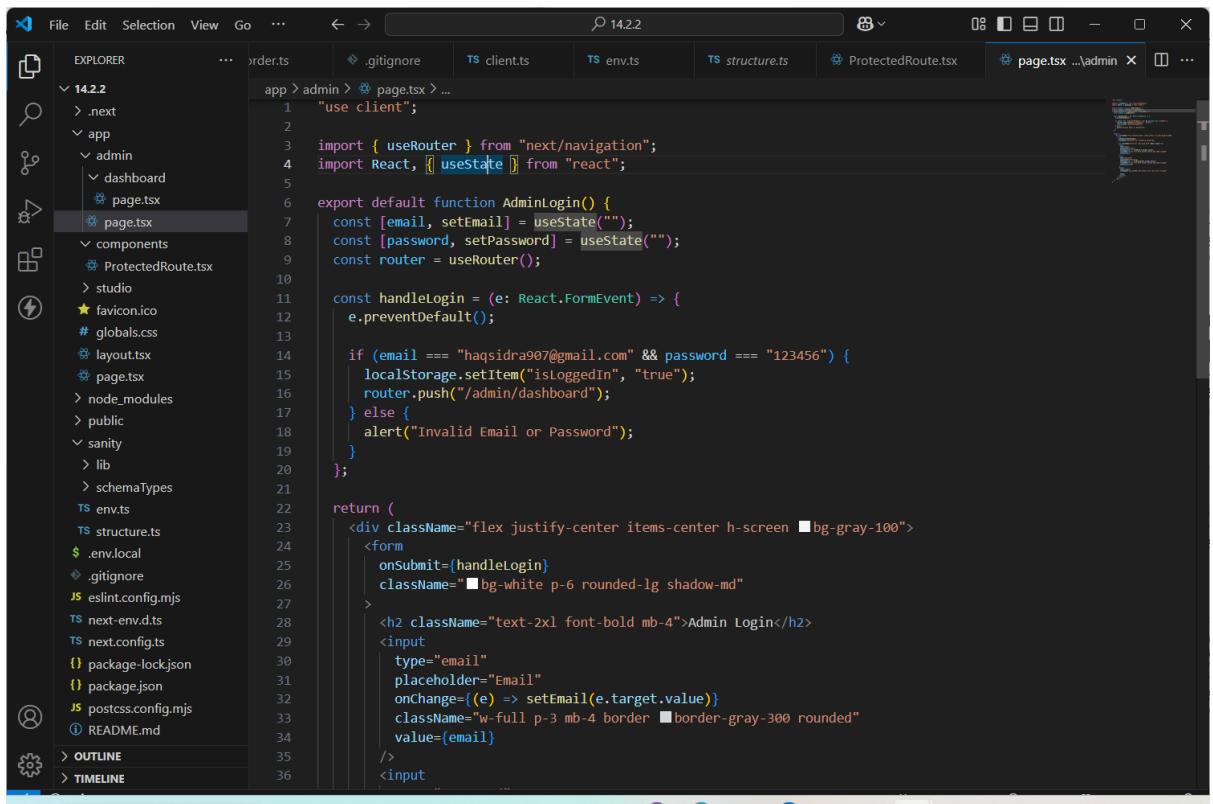
Screen Recording
2025-02-08 194936.m

Display Orders in the Admin Panel

Once orders are fetched, display them in a table format.

🔑 Authentication (Admin.)

- Admin logs in with a **hardcoded email/password**.
- On successful login, it saves a **session token** and redirects to **/admin/dashboard**.



The screenshot shows the VS Code interface with the AdminLogin component's code in the center editor tab. The code handles user input for email and password, checks if they match hardcoded values ('haqsidra907@gmail.com' and '123456'), and either logs the user in by setting a session token in localStorage or alerts the user if the credentials are invalid. It then redirects the user to the '/admin/dashboard' route.

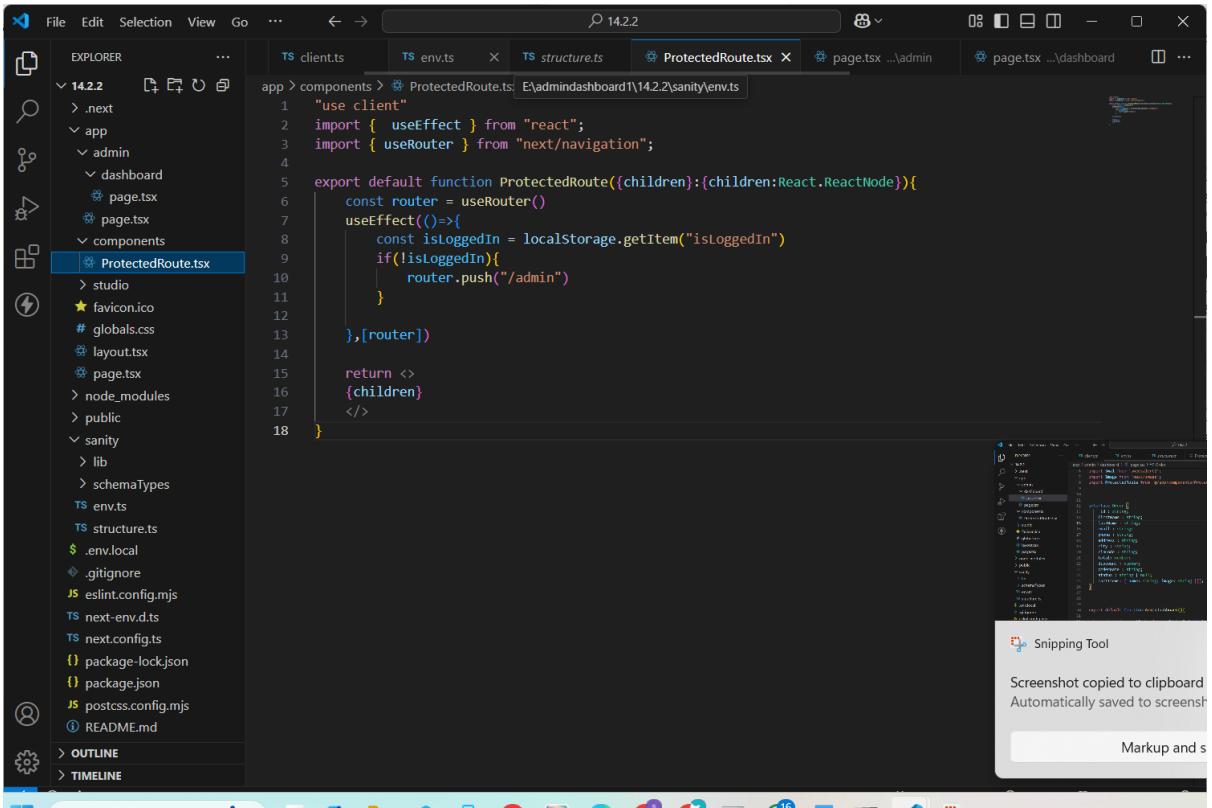
```
1  "use client";
2
3  import { useRouter } from "next/navigation";
4  import React, [ useState ] from "react";
5
6  export default function AdminLogin() {
7    const [email, setEmail] = useState("");
8    const [password, setPassword] = useState("");
9    const router = useRouter();
10
11   const handleLogin = (e: React.FormEvent) => {
12     e.preventDefault();
13
14     if (email === "haqsidra907@gmail.com" && password === "123456") {
15       localStorage.setItem("isLoggedIn", "true");
16       router.push("/admin/dashboard");
17     } else {
18       alert("Invalid Email or Password");
19     }
20
21   };
22
23   return (
24     <div className="flex justify-center items-center h-screen bg-gray-100">
25       <form
26         onSubmit={handleLogin}
27         className="bg-white p-6 rounded-lg shadow-md"
28       >
29         <h2 className="text-2xl font-bold mb-4">Admin Login</h2>
30         <input
31           type="email"
32           placeholder="Email"
33           onChange={(e) => setEmail(e.target.value)}
34           className="w-full p-3 mb-4 border border-gray-300 rounded"
35           value={email}
36         />
37         <input
38           type="password"
39           placeholder="Password"
40           onChange={(e) => setPassword(e.target.value)}
41           className="w-full p-3 mb-4 border border-gray-300 rounded"
42           value={password}
43         />
44       </form>
45     </div>
46   );
47 }
```



Protected Route (ProtectedRoute.tsx)

- Ensures only logged-in users can access the dashboard.

- Redirects unauthorized users back to the login page.



```

1  "use client"
2  import { useEffect } from "react";
3  import { useRouter } from "next/navigation";
4
5  export default function ProtectedRoute({children}:{children:React.ReactNode}){
6      const router = useRouter()
7      useEffect(()=>{
8          const isLoggedIn = localStorage.getItem("isLoggedIn")
9          if(!isLoggedIn){
10              router.push("/admin")
11          }
12      },[router])
13
14      return <>
15          {children}
16      </>
17  }
18

```

Fetch Orders (Dashboard.)

- Fetches orders from **Sanity CMS**.
- Displays a **table** with order details.
- Allows admin to **update order status** and **delete orders**.

```

File Edit Selection View Go ... ⏪ ⏩ 🔍 14.2.2 08 □ - ×
EXPLORER ... TS client.ts TS env.ts TS structure.ts ProtectedRoute.ts page.tsx ... \admin page.tsx ... \dashboard ...
14.2.2
> .next
app
  > admin
    > dashboard
      page.tsx
      page.tsx
      components
        ProtectedRoute.tsx
      studio
        favicon.ico
        # globals.css
        layout.tsx
        page.tsx
        node_modules
        public
        sanity
        lib
        schemaTypes
        TS env.ts
        TS structure.ts
        $ .env.local
        .gitignore
        JS eslint.config.mjs
        TS next-env.d.ts
        TS next.config.ts
        () package-lock.json
        () package.json
        JS postcss.config.mjs
        README.md
        OUTLINE
        TIMELINE
TS client.ts
TS env.ts
TS structure.ts
ProtectedRoute.tsx
page.tsx ... \admin
page.tsx ... \dashboard ...
1
2 import Swal from 'sweetalert2';
3 import Image from 'next/image';
4 import ProtectedRoute from '@/app/components/ProtectedRoute';
5
6 interface Order [
7   _id : string;
8   firstName : string;
9   lastName : string;
10  email : string;
11  phone : string;
12  address : string;
13  city : string;
14  zipcode : string;
15  total : number;
16  discount : number;
17  orderDate : string;
18  status : string | null;
19  cartItems: { name: string; image: string }[];
20
21 ]
22
23
24
25
26
27
28
29
30 export default function AdminDashboard(){
31
32   const [order ,setorders] = useState<Order[]>([])
33   const [selectedorderId, setSelectedorderId] = useState<string | null>(null)
34   const [filter, setFilter] = useState<string>("All")
35
36   useEffect(()=>{
37     client.fetch(`*[_type == "order"]{
38       _id,
39       firstName,
40       lastName,
41       email,

```

Order Actions

- **Update Status:** Admin can mark orders as **Pending, Dispatched, or Completed.**
- **Delete Order:** Admin can remove an order permanently

Screenshot of admin panel



Lighthouse Tool:

Npm install -g lighthouse

Lighthouse <http://localhost:3000> –view

☒ **Best Practices (100/100)** – Your website follows industry standards, ensuring security, modern coding techniques, and best development practices.

☒ **SEO (92/100)** – Your site is well-optimized for search engines, which helps improve visibility and reach more customers.

 **Good Accessibility (66/100)** – Your website is fairly accessible, meaning it's usable for a wide range of users, including those with disabilities.

Project Conclusion

This project successfully implements a **fully functional e-commerce admin dashboard** using **Next.js, TypeScript, Tailwind CSS, and Sanity CMS**. The application ensures a smooth and efficient order management system with **secure authentication, real-time order updates, and an intuitive UI**.

Key Achievements:

- **Secure Admin Authentication** – Only authorized users can access the dashboard.
- **Order Management System** – View, update, and delete customer orders dynamically.
- **Real-time Status Updates** – Orders can be marked as **Pending, Dispatched, or Completed**.
- **Sanity CMS Integration** – Orders are stored and managed through a headless CMS.
- **User-friendly Interface** – Responsive and optimized for seamless admin experience.

Future Enhancements:

- ◆ Implement **JWT authentication** for improved security.
- ◆ Add **role-based access control** for multiple admin levels.
- ◆ Enable **email notifications** for order status updates.
- ◆ Optimize data fetching with **GraphQL API**.
- ◆ Introduce **analytics & reporting** for better business insights.

Final Thoughts:

This project provides a **robust foundation for e-commerce order management**, making it easier for admins to track and process customer orders. By improving security, performance, and automation, this dashboard can be expanded into a **scalable and enterprise-level** admin panel.

| 1 | Test Case ID | Test Case | Test Steps | Expected Result | Actual Result | Status | Severity Level | Asg To | Remarks |
|---|--------------|---------------------------------|--|--|---------------------------------------|--------|----------------|--------|--------------------|
| 2 | TC001 | Validate product listing page | Open product page > Verify products | Products displayed correctly | Products displayed correctly | Passed | High | - | No issues found |
| 3 | TC002 | Test API error handling | Disconnect API > Refresh page | Show fallback UI with error message | Error message shown | Passed | Medium | - | Handled gracefully |
| 4 | TC003 | Check cart functionality | Add product to cart > Verify cart contents | Cart updates with added product | Cart updates as expected | Passed | High | - | Works as expected |
| 5 | TC004 | Ensure responsiveness on mobile | Resize browser window > Check layout | Layout adjusts properly to screen size | Responsive layout working as intended | Passed | Medium | - | Test successful |

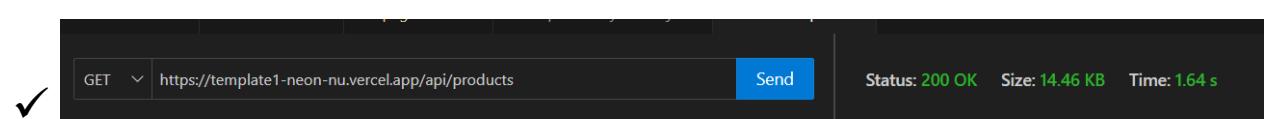
✓ Tools: Thunder Client (I used Thunder Client to get data)

The screenshot shows the Thunder Client interface with a successful API request. The request URL is `https://template1-neon-nu.vercel.app/api/products`. The response status is `200 OK`, size is `14.46 KB`, and time is `1.64 s`. The response body is a JSON object:

```

1  [
2   {
3     "imageUrl": "https://cdn.sanity.io/images/7xt4qcah/production/4e2ed6a9ea6e1413843e5f3113cf210dc301-278x296.png",
4     "discountPercent": 20,
5     "isNew": true,
6     "colors": [
7       "Blue",
8       "Red",
9       "Black",
10      "Yellow"
11    ],
12    "sizes": [
13      "S",
14      "XXL",
15      "XL",
16      "L"
17    ],
18    "_id": "edc7cb47-8599-45d0-b02c-34429f7a639e",
19    "description": "This stylish green bomber jacket offers a sleek and modern twist on a classic design. Made from soft and comfortable fabric, it features snap buttons and ribbed cuffs, giving it a sporty yet refined look. The minimalist style makes it perfect for layering over casual t-shirts or hoodies. Whether you're out with friends or just lounging, this jacket provides a laid-back yet fashionable vibe. Its muted green color adds a subtle, earthy tone to any outfit.", 
20    "category": "Outerwear", 
21    "name": "Casual Bomber Jacket", 
22    "price": 30
  
```

A note at the bottom right of the response pane states: "⚠ This is the non-commercial version of Thunder Client. For business or professional use, please consider purchasing a license." A "View Terms" link is also present.



Share your experience

- We faced multiple hurdles, especially in **dynamic routing, data fetching, and optimizing performance** using tools like **Lighthouse**.
- **Deployment Issues:** Resolved environment-specific errors . by properly handling .env files.

Additional Comments;

- Working on this e-commerce website has been an incredible learning journey. Since this was my first project of this scale, I faced many challenges but also gained valuable experience along the way.
- Optimized images using Next.js <Image> component and **lazy loading**.
- Learned about **protecting API routes and handling authentication tokens properly**.
- Error handling (404 pages, API failures, loading states) improves UX
- Faced several **deployment issues** while pushing the project to **Vercel**.
- Learned how to properly configure **environment variables (.env.local)** to avoid errors.