Day 3 - API Integration Report – Robiz

Table of Context:

- API integration process
- Adjustments made to schemas
- Screenshots of:
    1. API calls.
    2. Data successfully displayed in the frontend.
    3. Populated Sanity CMS fields.

- Code snippets for API integration and migration scripts

1. API integration process:

Overview

This report documents the process of integrating APIs for fetching and displaying product data in a Next.js application using Sanity CMS. The data was manually added to Sanity for better control and customization, ensuring alignment with project requirements. The integration involves querying data using GROQ, serving it through API routes, and consuming it in React components using useEffect.

1. Data Management in Sanity

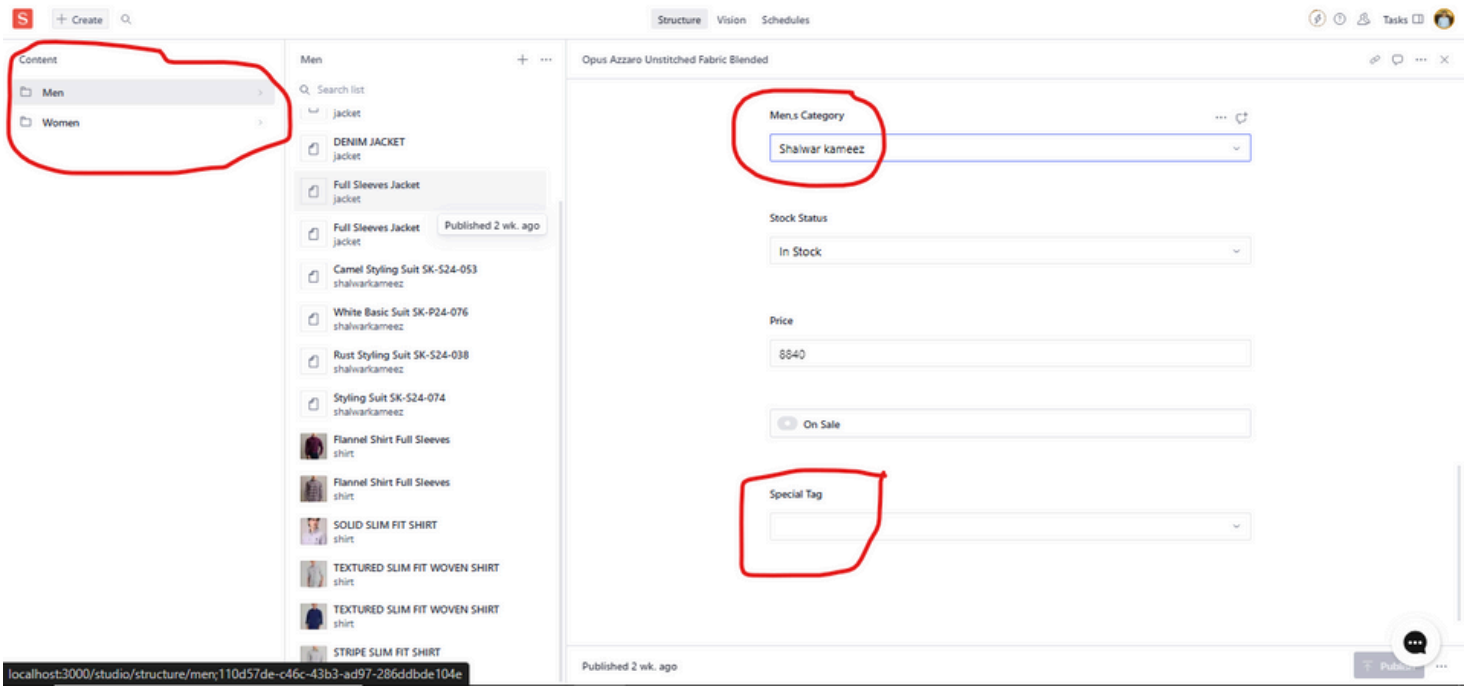
Data was manually added to Sanity Studio to meet specific requirements. This approach provided the flexibility to organize the data precisely as needed. Categories and subcategories of products included:

Categories:

- Men
  - Subcategories: Shirt, Pant, T-shirt
- Women
  - Subcategories: Tops, Dress, Frock

Tags:

- New Arrival
- Special Offer
- Featured
- Top trending



2. API Routes

Custom API routes were created in the Next.js application to fetch data from Sanity CMS. The routes serve as endpoints for querying specific product categories, subcategories, or individual product details.

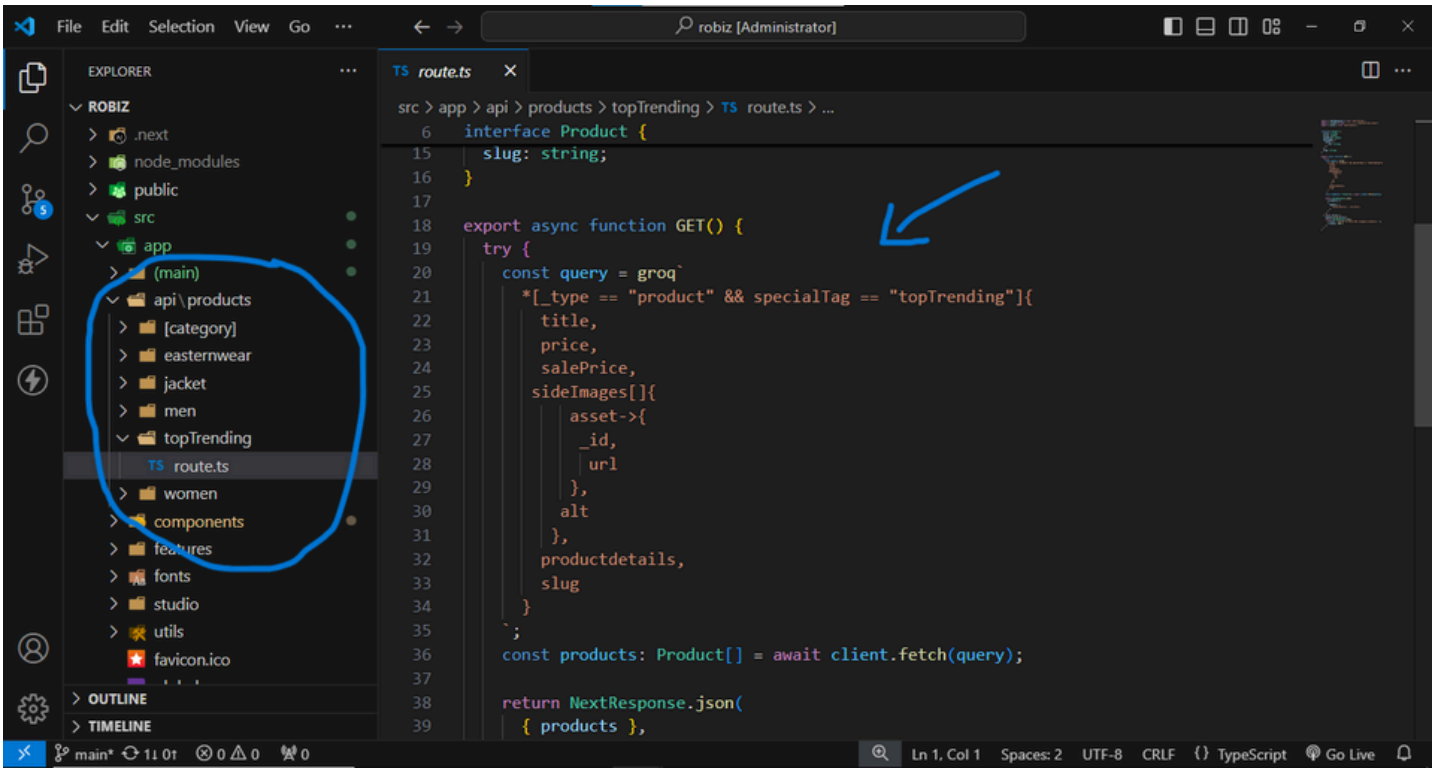
Example API Endpoints:

1. All Easternwear Products:
   /api/products/easternwear

2. Category and Subcategory Based:
   /api/products/[category]/[subcategory]

3. *Signle product detail page*:
   /api/products/[category]/[subcategory]/[product]

4. Tagged Products: */api/products/[tag]*

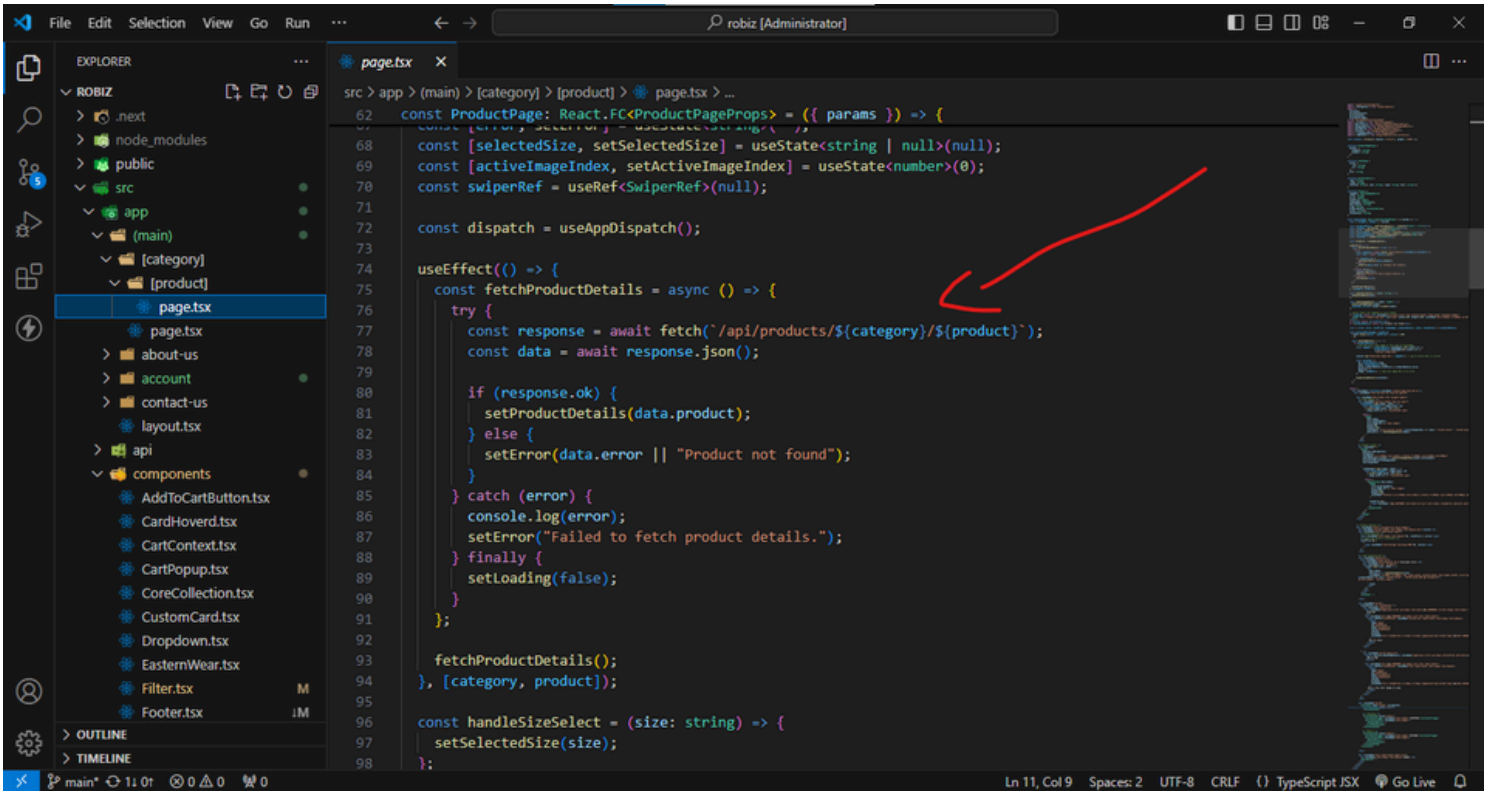


Fetching Data in Components:

The useEffect hook is used in React components to fetch data from the API endpoints. Data is dynamically retrieved and displayed to the user. This approach ensures the UI updates automatically when the data changes.

Step-by-Step Explanation

- Initialize State: Create state variables for storing products and loading status.
- Define Fetch Function: Use fetch to call the API endpoint and retrieve the data.
- Handle Loading and Errors: Display loading indicators while data is being fetched and handle any errors that occur.
- Update State: Populate the state with the fetched data to render it in the UI.



and

```tsx
const [selectedSize, setSelectedSize] = useState<string | null>(null);
const [activeImageIndex, setActiveImageIndex] = useState<number>(0);
const swiperRef = useRef<SwiperRef>(null);

const dispatch = useAppDispatch();

useEffect(() => {
  const fetchProductDetails = async () => {
    try {
      const response = await fetch(`/api/products/${category}/${product}`);
      const data = await response.json();

      if (response.ok) {
        setProductDetails(data.product);
      } else {
        setError(data.error || "Product not found");
      }
    } catch (error) {
      console.log(error);
      setError("Failed to fetch product details.");
    } finally {
      setLoading(false);
    }
  };

  fetchProductDetails();
}, [category, product]);
```

API calls:

I called api from api routes , here you see how i called apis in frontend components and pages

app/[category]/[product]

```
useEffect(() => {
  const fetchProductDetails = async () => {
    try {
      const response = await fetch(`/api/products/${category}/${product}`);
      const data = await response.json();

      if (response.ok) {
        setProductDetails(data.product);
      } else {
        setError(data.error || "Product not found");
      }
    } catch (error) {
      console.log(error);
      setError("Failed to fetch product details.");
    } finally {
      setLoading(false);
    }
  };

  fetchProductDetails();
}, [category, product]);
```

app/[category]

```
const fetchProducts = async () => {
  try {
    const response = await fetch(`/api/products/${params.category}`);
    const data = await response.json();
    if (response.ok) {
      setProducts(data.products);
      setSortedProducts(data.products);
    } else {
      setError(data.error || "Failed to fetch products");
    }
  } catch (err) {
    console.log(err);
    setError("Failed to fetch products");
  } finally {
    setLoading(false);
  }
};

fetchProducts();
}, [params.category]);
```

easterwear

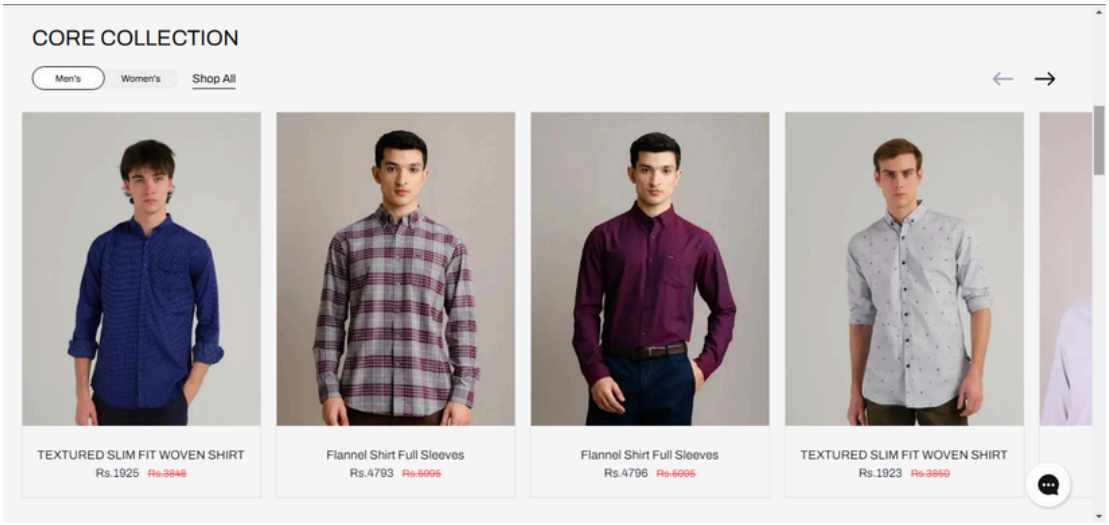
```
useEffect(() => {
    const fetchProducts = async () => {
        setLoading(true);
        try {
            const response = await fetch(`/api/products/easternwear`);
            const data = await response.json();
            setProducts(data.products);
        } catch (error) {
            console.error("Failed to fetch products:", error);
        } finally {
            setLoading(false);
        }
    };

    fetchProducts();
}, []);
```

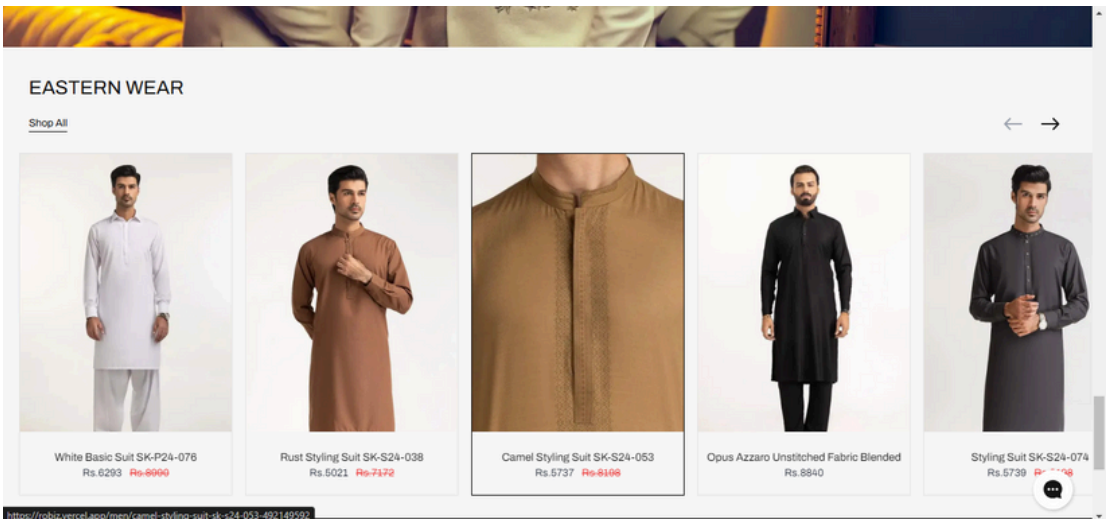
The use of API routes for fetching and processing data from external sources, followed by the integration of the data into Sanity CMS, significantly improves the application's architecture, security, and performance. It ensures that the frontend receives clean, structured data, while the backend handles business logic and content management efficiently.

Data successfully displayed in the frontend:

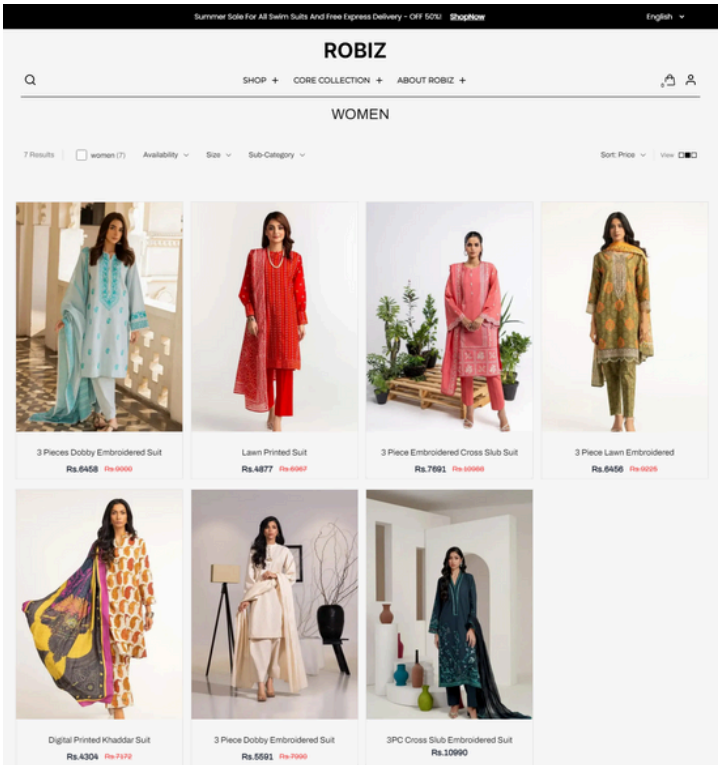
After integrating and processing the data from the API through API routes, the next key objective was to ensure that the data was successfully displayed in the frontend of the application.

This step is critical to ensure that the user interface (UI) presents accurate and up-to-date information from the backend or external sources.





Category page



Product page