***DAY 03: “API INTEGRATION AND DATA MIGRATION***

***Report”***

***General e-commerce website of Comforty Chair***

**API integration process:**

Integrating an API into your Next.js e-commerce site can enhance its functionality, whether it's for managing products, processing payments, or handling user authentication. Here's a general process to integrate an API:

**1. Choose the API**

Decide which API you want to integrate, such as:

* **Product data** (e.g., external product inventory API)
* **Payment gateways** (e.g., Stripe, PayPal)
* **User authentication** (e.g., Firebase Auth, Auth0)
* **Shipping APIs** (e.g., Shippo, USPS)
* **CMS** (e.g., Contentful, Sanity for content management)

**2. Setup API Keys**

Most APIs require an API key or secret for authentication. Follow the service's documentation to obtain these credentials.

* Example for Stripe:

javascript

CopyEdit

const stripe = require('stripe')(process.env.STRIPE\_SECRET\_KEY);

**3. Create API Routes in Next.js**

Next.js allows you to create server-side API routes in the pages/api directory. These routes act as backend endpoints for handling API requests.

Example of a basic Next.js API route:

javascript

CopyEdit

// pages/api/products.js

export default async function handler(req, res) {

if (req.method === 'GET') {

const response = await fetch('https://externalapi.com/products');

const products = await response.json();

res.status(200).json(products);

} else {

res.status(405).end(); // Method Not Allowed

}

}

**4. Fetching Data from the API**

Use the API route you created to fetch data. You can use fetch or Axios in your components to call your API route.

Example for a component fetching data:

javascript

CopyEdit

// pages/index.js

import { useEffect, useState } from 'react';

export default function Home() {

const [products, setProducts] = useState([]);

useEffect(() => {

async function getProducts() {

const response = await fetch('/api/products');

const data = await response.json();

setProducts(data);

}

getProducts();

}, []);

return (

<div>

<h1>Our Products</h1>

<ul>

{products.map((product) => (

<li key={product.id}>{product.name}</li>

))}

</ul>

</div>

);

}

**5. Handling Authentication (Optional)**

If your API involves user authentication (e.g., login or signup), use libraries like next-auth or firebase-admin for authentication flow.

Example using next-auth:

javascript

CopyEdit

import { signIn, signOut, useSession } from 'next-auth/react';

function LoginPage() {

const { data: session } = useSession();

if (session) {

return (

<>

<p>Welcome, {session.user.name}!</p>

<button onClick={() => signOut()}>Sign Out</button>

</>

);

}

return (

<button onClick={() => signIn('google')}>Sign In with Google</button>

);

}

export default LoginPage;

**6. Handling Payment Integration (Optional)**

If integrating a payment gateway like Stripe, you’ll typically have both client-side (to capture payment) and server-side (to handle API calls to Stripe) components.

Example using Stripe:

javascript

CopyEdit

// pages/api/create-checkout-session.js

import Stripe from 'stripe';

const stripe = new Stripe(process.env.STRIPE\_SECRET\_KEY);

export default async function handler(req, res) {

if (req.method === 'POST') {

const session = await stripe.checkout.sessions.create({

payment\_method\_types: ['card'],

line\_items: req.body.items.map(item => ({

price\_data: {

currency: 'usd',

product\_data: {

name: item.name,

},

unit\_amount: item.price \* 100,

},

quantity: item.quantity,

})),

mode: 'payment',

success\_url: `${process.env.NEXT\_PUBLIC\_SITE\_URL}/success`,

cancel\_url: `${process.env.NEXT\_PUBLIC\_SITE\_URL}/cancel`,

});

res.status(200).json({ id: session.id });

}

}

**7. Test and Debug**

Ensure to test the API integration thoroughly in both development and production environments.

**8. Deploy and Monitor**

Deploy your Next.js app (e.g., on Vercel) and monitor API requests for performance and reliability.

***Adjustments made to Schemas:***

When you make adjustments to the schemas in your e-commerce application, it typically involves modifying the data models to match new requirements, improving performance, or adding new features. Whether you're using a database like MongoDB, SQL, or a headless CMS, the process for adjusting the schema will vary slightly. Below is a general approach based on common scenarios.

### Steps to Adjust Schemas:

#### 1. **Identify Schema Changes**

* **Add Fields**: New properties or attributes need to be added (e.g., discountPrice, inventoryCount).
* **Modify Fields**: Changing the data type or constraints of existing fields (e.g., changing price from a string to a float).
* **Remove Fields**: Removing unnecessary or deprecated fields (e.g., removing oldCategory if no longer needed).
* **Relationships**: Adjusting relationships, such as adding a foreign key or adjusting a reference.

#### 2. **Adjusting Database Schema (Relational Database)**

If you're using an SQL-based database (like MySQL, PostgreSQL), you need to modify your schema using migration scripts or queries.

* **Example: Adding a New Field** in a Product Table

sql

CopyEdit

ALTER TABLE products ADD COLUMN discountPrice DECIMAL(10, 2);

* **Example: Changing Data Type**

sql

CopyEdit

ALTER TABLE products MODIFY price DECIMAL(10, 2);

* **Example: Removing a Column**

sql

CopyEdit

ALTER TABLE products DROP COLUMN oldCategory;

If you’re using **Prisma** or **Sequelize** ORM in your Next.js app, you can update the models and run migrations to adjust the schema:

* **Prisma Example**: Modify your Prisma schema file (typically schema.prisma):

prisma

CopyEdit

model Product {

id Int @id @default(autoincrement())

name String

price Float

discountPrice Float? // Optional field for discounted price

category String

}

Then run the migration command to apply the changes:

bash

CopyEdit

npx prisma migrate dev --name add\_discount\_price

#### 3. **Adjusting NoSQL Schema (MongoDB)**

MongoDB is schema-less, but it's good practice to maintain a consistent structure. If you're using an ODM (e.g., **Mongoose** for Node.js), you'll need to update the schema definition and re-deploy your app.

* **Example: Adding a New Field in Mongoose**

javascript

CopyEdit

const productSchema = new mongoose.Schema({

name: { type: String, required: true },

price: { type: Number, required: true },

discountPrice: { type: Number, default: 0 }, // New field

category: { type: String, required: true }

});

const Product = mongoose.model('Product', productSchema);

* **Modifying Schema**: For example, if you want to change the price to store it as a Decimal instead of a Number for better precision:

javascript

CopyEdit

productSchema.path('price').set(value => parseFloat(value));

After updating the schema, ensure that your database can handle existing records, especially when adding or removing fields.

#### 4. **Updating API Routes or GraphQL Schema**

After making changes to your database schema, you need to reflect these changes in your API layer.

* **REST API (Next.js API Routes)**: If you're using Next.js API routes, update the handler code to reflect the new schema changes (e.g., reading the new discountPrice field).

javascript

CopyEdit

// pages/api/products.js

export default async function handler(req, res) {

if (req.method === 'GET') {

const products = await db.product.findMany(); // Adjusted to fetch the new field

res.status(200).json(products);

}

}

* **GraphQL API**: If using GraphQL, adjust your GraphQL schema definitions to include the new or modified fields:

graphql

CopyEdit

type Product {

id: ID!

name: String!

price: Float!

discountPrice: Float // New field

category: String!

}

type Query {

products: [Product]!

}

#### 5. **Update Frontend to Reflect Schema Changes**

Your frontend (Next.js) needs to be updated to handle any changes in the schema, such as displaying new fields or adjusting how data is fetched.

* **Adjust Components**: For instance, if you added a discountPrice field, update the components that render product data:

javascript

CopyEdit

export default function ProductList({ products }) {

return (

<div>

{products.map(product => (

<div key={product.id}>

<h3>{product.name}</h3>

<p>Price: ${product.price}</p>

{product.discountPrice > 0 && <p>Discounted Price: ${product.discountPrice}</p>}

</div>

))}

</div>

);

}

* **Fetch the Updated Data**: If the schema changed and you’re fetching new fields (e.g., discountPrice), make sure your API calls or GraphQL queries reflect those changes:

javascript

CopyEdit

useEffect(() => {

async function fetchProducts() {

const res = await fetch('/api/products');

const data = await res.json();

setProducts(data);

}

fetchProducts();

}, []);

#### 6. **Data Migration (Optional)**

If you’re removing or changing important fields in the schema (e.g., data type changes), you may need to run a migration to update the existing data. This step is especially important if you are working with an existing production database.

* **Example Migration Script** (MongoDB):

javascript

CopyEdit

async function migrateData() {

const products = await Product.find({});

for (let product of products) {

// Migrate old data to new structure

if (product.oldField) {

product.newField = product.oldField;

await product.save();

}

}

}

***Migration steps and tools used:***

Migration is a crucial process for adjusting your database schema in a live environment, ensuring that new changes are applied smoothly without disrupting data or functionality. Depending on whether you're using a relational database (SQL) or a NoSQL database (e.g., MongoDB), migration tools and techniques will differ. Here's a general overview of migration steps and the tools commonly used:

**Steps for Migration:**

1. **Plan the Schema Changes**
   * Understand and plan the changes you want to make to your schema, such as adding new columns, modifying existing ones, or removing old fields.
   * Identify whether these changes require data migration (i.e., changing the existing records to fit the new schema).
2. **Backup the Database**
   * Before making any changes, **back up** your database to avoid potential data loss or corruption.
   * Tools like **pg\_dump** (PostgreSQL), **mysqldump** (MySQL), or MongoDB's **mongodump** are commonly used for backups.
3. **Create Migration Files**
   * Migration files contain the SQL or script code needed to apply the changes (e.g., ALTER TABLE statements in SQL, field adjustments in MongoDB).
   * These files typically contain both **up** migrations (applying changes) and **down** migrations (reverting changes).
4. **Apply Migrations**
   * Use migration tools to apply the changes to the database.
   * Ensure that you apply migrations incrementally, especially in a production environment, to avoid downtime or data corruption.
5. **Test the Migration Locally**
   * Test the migration on a **staging or local environment** to ensure that all changes are applied as expected.
   * Run the migration scripts and test both the database schema and application functionality (e.g., API endpoints, UI components).
6. **Deploy to Production**
   * Once testing is successful, apply the migrations to the production environment.
   * Monitor for any issues post-deployment and ensure that the system is stable.

Data Migration Options: To streamline your migration process, here are three methods you can use. Ensure you validate all data and document your migration steps. Using the Provided API: Write scripts to fetch and transform data from the API (You can find the right script in the above-mentioned template, and you can use any template data depending on your flexibility.).

Manual Import:

• Export data from the API or another source as JSON or CSV.

• Use Sanity’s built-in import tools to upload the data.

• This method is ideal for small datasets or when learning basic import methods. Using External Platform APIs:

• Fetch data from platforms like Shopify or WooCommerce.

• Map fields to your Sanity schema.

• Follow a similar script-based migration process, ensuring data transformation aligns with schema requirements.

### Data Migration Script:

The migration script is available in the project repository:

[Migration Script](https://www.google.com/url?q=https://www.google.com/url?q%3Dhttps://github.com/Hamzah-syed/giaic-hackathon-template-08/blob/master/scripts/migrate.mjs%26amp;sa%3DD%26amp;source%3Deditors%26amp;ust%3D1737225059989438%26amp;usg%3DAOvVaw3pBGjqGlJAoWP3nFqgZ1aH&sa=D&source=docs&ust=1737225059998715&usg=AOvVaw1zqriKUO1LSWr-S00YKWST)

This script automates the process of transferring data from the provided REST APIs to your Sanity dataset.

#### **Setting Up Environment Variables**

1. Create a .env file in the root of your project. **Do not depend on .env.local file for the script, you need to create .env file to make the script work**

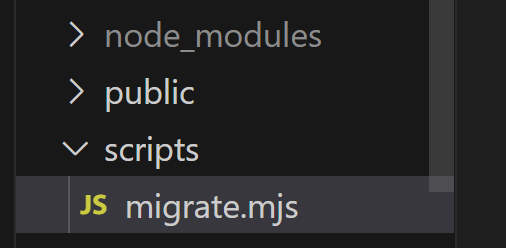
Create .env file and add the following variables:

NEXT\_PUBLIC\_SANITY\_PROJECT\_ID="<Project ID>" # Add your project Id

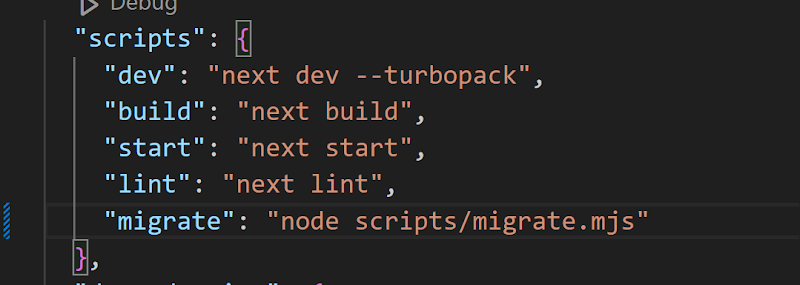
NEXT\_PUBLIC\_SANITY\_DATASET="production"

NEXT\_PUBLIC\_SANITY\_AUTH\_TOKEN="<Auth Token>" # Add your token

1. Refer to the Practice Hackathon Docs for details on how to retrieve these values.
2. Create migrate.mjs inside of the script folder
3. Get the migration code from [here](https://www.google.com/url?q=https://www.google.com/url?q%3Dhttps://github.com/Hamzah-syed/giaic-hackathon-template-08/blob/master/scripts/migrate.mjs%26amp;sa%3DD%26amp;source%3Deditors%26amp;ust%3D1737225059990431%26amp;usg%3DAOvVaw2l1rqqVjw0LRkYfftq6vYa&sa=D&source=docs&ust=1737225059999194&usg=AOvVaw2ftoDUfoipFM6MtQ1QMIup)



1. Open `package.json` file and add the following code inside of scripts: "migrate": "node scripts/migrate.mjs"



1. Install the following package before running the script
2. npm install dotenv
3. Now run the command npm run migrate
4. This will insert the data from the rest api to your sanity studio.

### Rest API endpoint for Details:

### Using these API ;

#### Products API

* Endpoint: [https://giaic-hackathon-template-08.vercel.app/api/products](https://www.google.com/url?q=https://www.google.com/url?q%3Dhttps://giaic-hackathon-template-08.vercel.app/api/products%26amp;sa%3DD%26amp;source%3Deditors%26amp;ust%3D1737225059991225%26amp;usg%3DAOvVaw0s7jVC7JeYpLit_7lFAybt&sa=D&source=docs&ust=1737225059999590&usg=AOvVaw1ZaUtW10Gj634bkB35YTcv)

**Categories API:**

* Endpoint: [https://giaic-hackathon-template-08.vercel.app/api/categories](https://www.google.com/url?q=https://www.google.com/url?q%3Dhttps://giaic-hackathon-template-08.vercel.app/api/categories%26amp;sa%3DD%26amp;source%3Deditors%26amp;ust%3D1737225059991520%26amp;usg%3DAOvVaw3IHpjboALEwVNLDmsU00LM&sa=D&source=docs&ust=1737225059999747&usg=AOvVaw333Fv5FGo45Dsv8oHfXqAi)