API Integration and Data Migration Report - Day 3

Executive Summary

On Day 3 of the Hekto Marketplace Builder Hackathon 2025, the focus was on API integration, data migration, and error handling within a Next.js application. The tasks included understanding the API, adjusting schemas, migrating data, and integrating APIs into the Next.js project. This report summarizes the accomplishments and outlines the steps taken.

API Integration

API Understanding and Integration Process

- **API Integration Report:** Focused on integrating APIs for the FURNIRO project.
- **Server and Database:** Detailed the integration process involving server and database interactions.

API Details

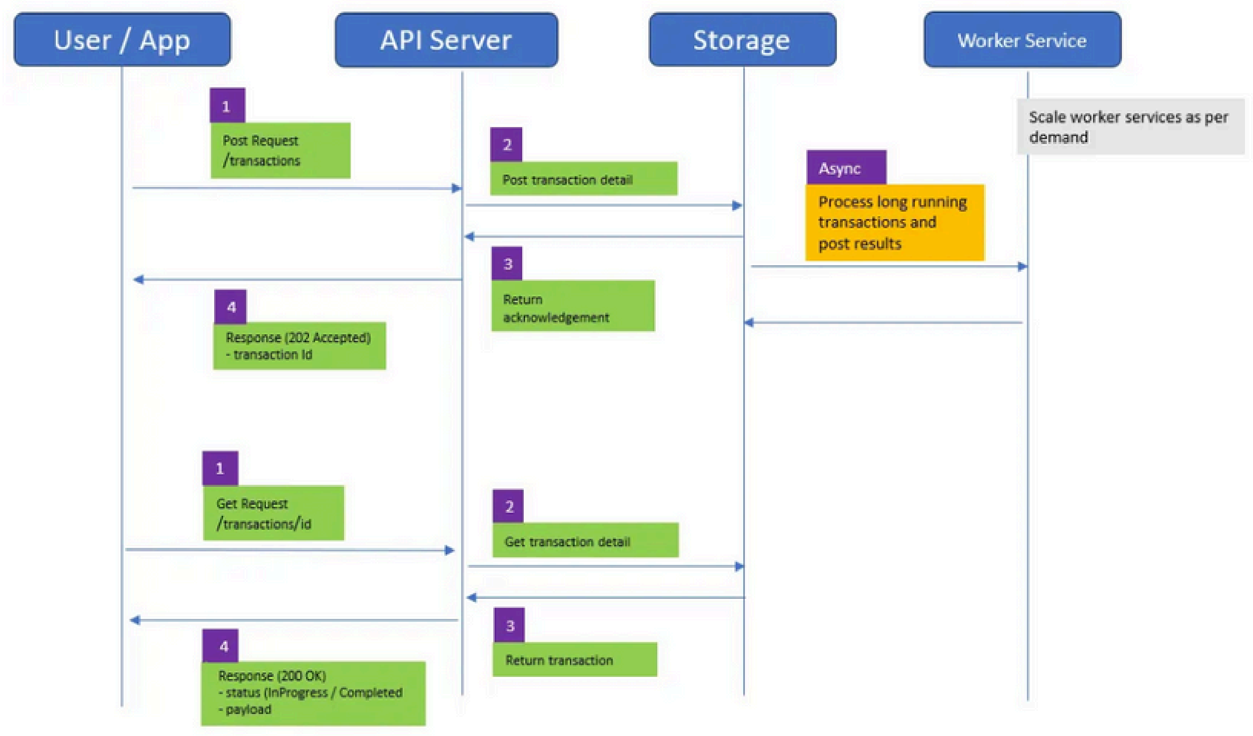
- **Endpoints Used:** [List of endpoints]
- **Data Formats:** [JSON, XML, etc.]
- **Integration Steps:**
    1. Established connections with the server and database.
    2. Implemented API calls to fetch and post data.

3. Resolved challenges such as rate limiting and data consistency.

Challenges and Solutions

- **Challenge:** API rate limiting.
- **Solution:** Implemented caching and pagination.

API Integration Process Flow


User / App
API Server
Storage
Worker Service
Scale worker services as per demand
1
Post Request /transactions
2
Post transaction detail
Async
Process long running transactions and post results
3
Return acknowledgement
4
Response (202 Accepted) - transaction Id
1
Get Request /transactions/id
2
Get transaction detail
3
Return transaction
4
Response (200 OK) - status (InProgress / Completed - payload

Let me explain each component and its role in the API integration process:

Browser/Client

- Initiates the API requests

- Handles user interactions

- Displays the received data

- Manages the user interface

Internet

- Acts as the communication medium
- Handles request/response routing
- Ensures data transmission between client and API

API Layer

- Sets rules for interaction
- Manages data transfer

- Handles:

- Authentication
- Authorization
- Rate limiting
- Request validation
- Response formatting
- Error handling

Server

- Processes business logic
- Handles API requests
- Manages application state
- Coordinates with the database

- Implements security measures
- Performs data transformations

Database

- Stores application data
- Handles data persistence
- Manages data relationships
- Ensures data integrity
- Provides data querying capabilities

The process flow works as follows:

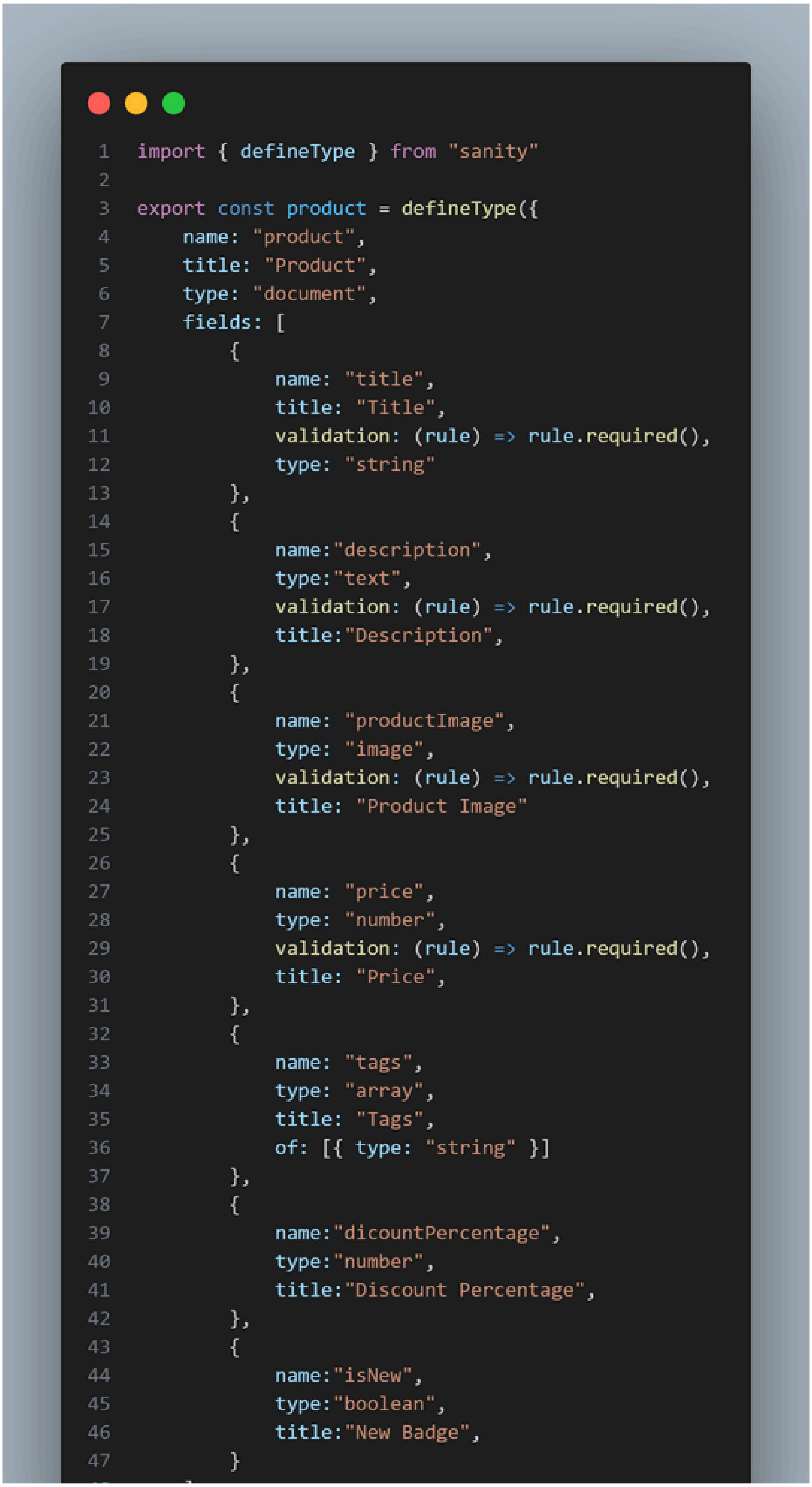
1. The browser sends an HTTP request to the API
2. The request travels through the internet to reach the API layer
3. The API layer validates and processes the request
4. The server receives the processed request and performs necessary operations
5. The server queries the database if data is needed
6. The database returns the requested data to the server
7. The server processes the data and sends it back through the API layer
8. The API formats the response and sends it back to the browser
9. The browser receives and displays the data

This integration process ensures:

- Secure data transmission
- Proper data validation
- Efficient request handling
- Scalable architecture
- Separation of concerns
- Standardized communication

Schemas Adjustments

Product.ts



```
import { defineType } from "sanity"

export const product = defineType({
    name: "product",
    title: "Product",
    type: "document",
    fields: [
        {
            name: "title",
            title: "Title",
            validation: (rule) => rule.required(),
            type: "string"
        },
        {
            name:"description",
            type:"text",
            validation: (rule) => rule.required(),
            title:"Description",
        },
        {
            name: "productImage",
            type: "image",
            validation: (rule) => rule.required(),
            title: "Product Image"
        },
        {
            name: "price",
            type: "number",
            validation: (rule) => rule.required(),
            title: "Price",
        },
        {
            name: "tags",
            type: "array",
            title: "Tags",
            of: [{ type: "string" }]
        },
        {
            name:"dicountPercentage",
            type:"number",
            title:"Discount Percentage",
        },
        {
            name:"isNew",
            type:"boolean",
            title:"New Badge",
        }
```

```
    ]
})
```

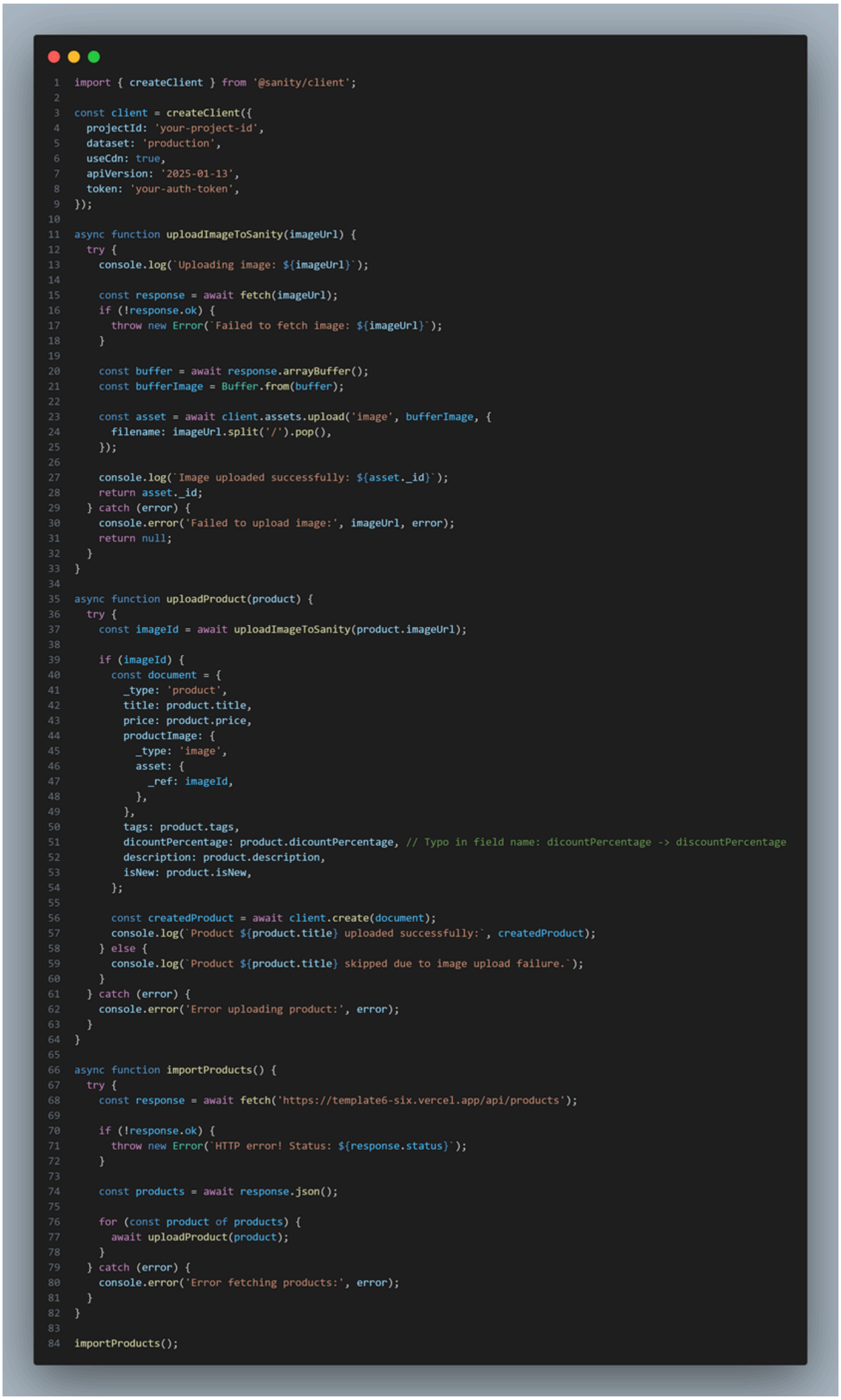
- **Schema Adjustments:** Made changes to schemas in D4V-2 product.ts and D4V-3 product.ts to ensure compatibility and efficiency.

Adjustments in D4V-2 product.ts and D4V-3 product.ts

- **Changes Made:**
  - Added new fields for product categories.
  - Modified data types for price fields to handle decimals.
- **Impact:**
  - Improved data accuracy and application functionality.

Migration Steps

script/importData.js



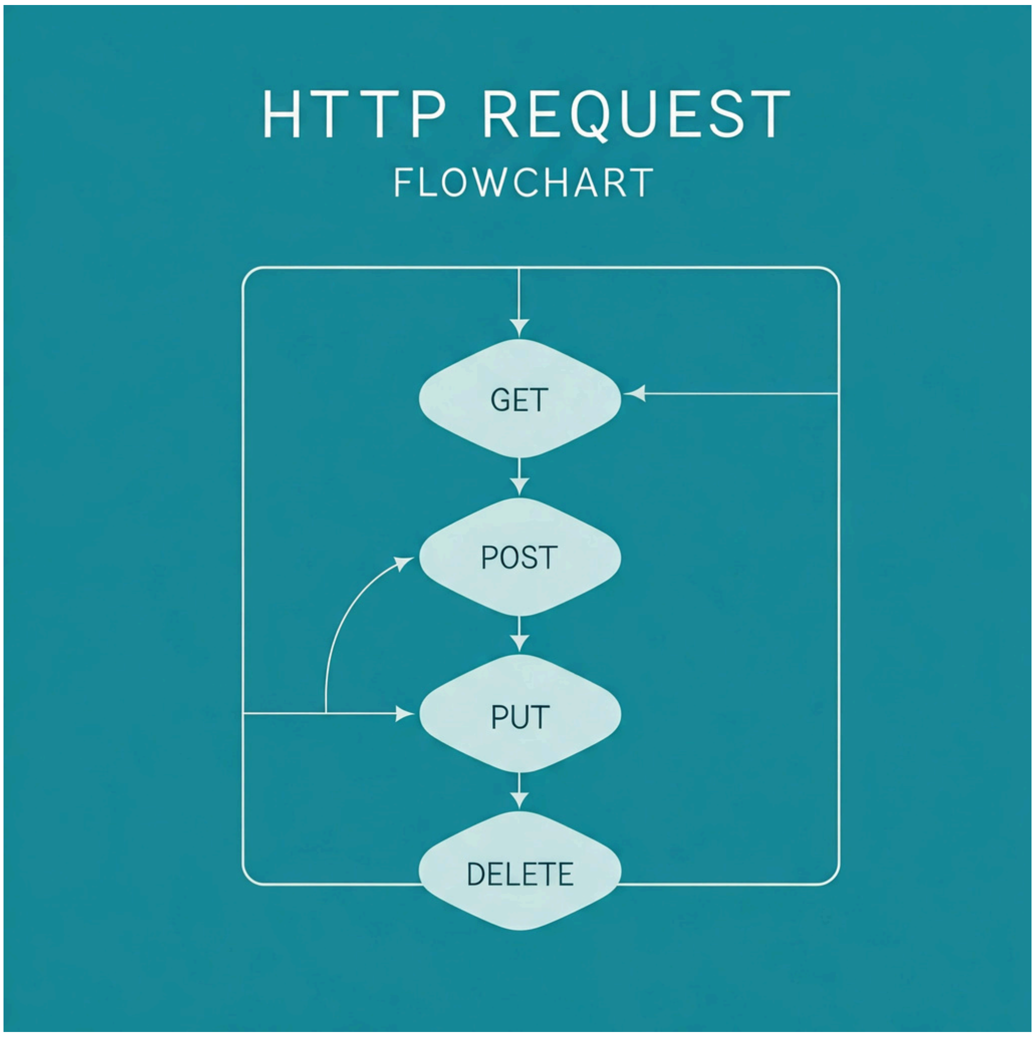
```javascript
import { createClient } from '@sanity/client';

const client = createClient({
  projectId: 'your-project-id',
  dataset: 'production',
  useCdn: true,
  apiVersion: '2025-01-13',
  token: 'your-auth-token',
});

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(),
    });

    console.log(`Image uploaded successfully: ${asset._id}`);
    return asset._id;
  } catch (error) {
    console.error('Failed to upload image:', imageUrl, error);
    return null;
  }
}

async function uploadProduct(product) {
  try {
    const imageId = await uploadImageToSanity(product.imageUrl);

    if (imageId) {
      const document = {
        _type: 'product',
        title: product.title,
        price: product.price,
        productImage: {
          _type: 'image',
          asset: {
            _ref: imageId,
          },
        },
        tags: product.tags,
        dicountPercentage: product.dicountPercentage, // Typo in field name: dicountPercentage -> discountPercentage
        description: product.description,
        isNew: product.isNew,
      };

      const createdProduct = await client.create(document);
      console.log(`Product ${product.title} uploaded successfully:`, createdProduct);
    } else {
      console.log(`Product ${product.title} skipped due to image upload failure.`);
    }
  } catch (error) {
    console.error('Error uploading product:', error);
  }
}

async function importProducts() {
  try {
    const response = await fetch('https://template6-six.vercel.app/api/products');

    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }

    const products = await response.json();

    for (const product of products) {
      await uploadProduct(product);
    }
  } catch (error) {
    console.error('Error fetching products:', error);
  }
}

importProducts();
```

- **Data Migration Method:** Utilized the provided API to migrate data.
- **Fetching and Transformation:** Employed the fetch method to retrieve and transform data.
- **Template and Validation:** Used templates for data import and implemented validation with error handling.

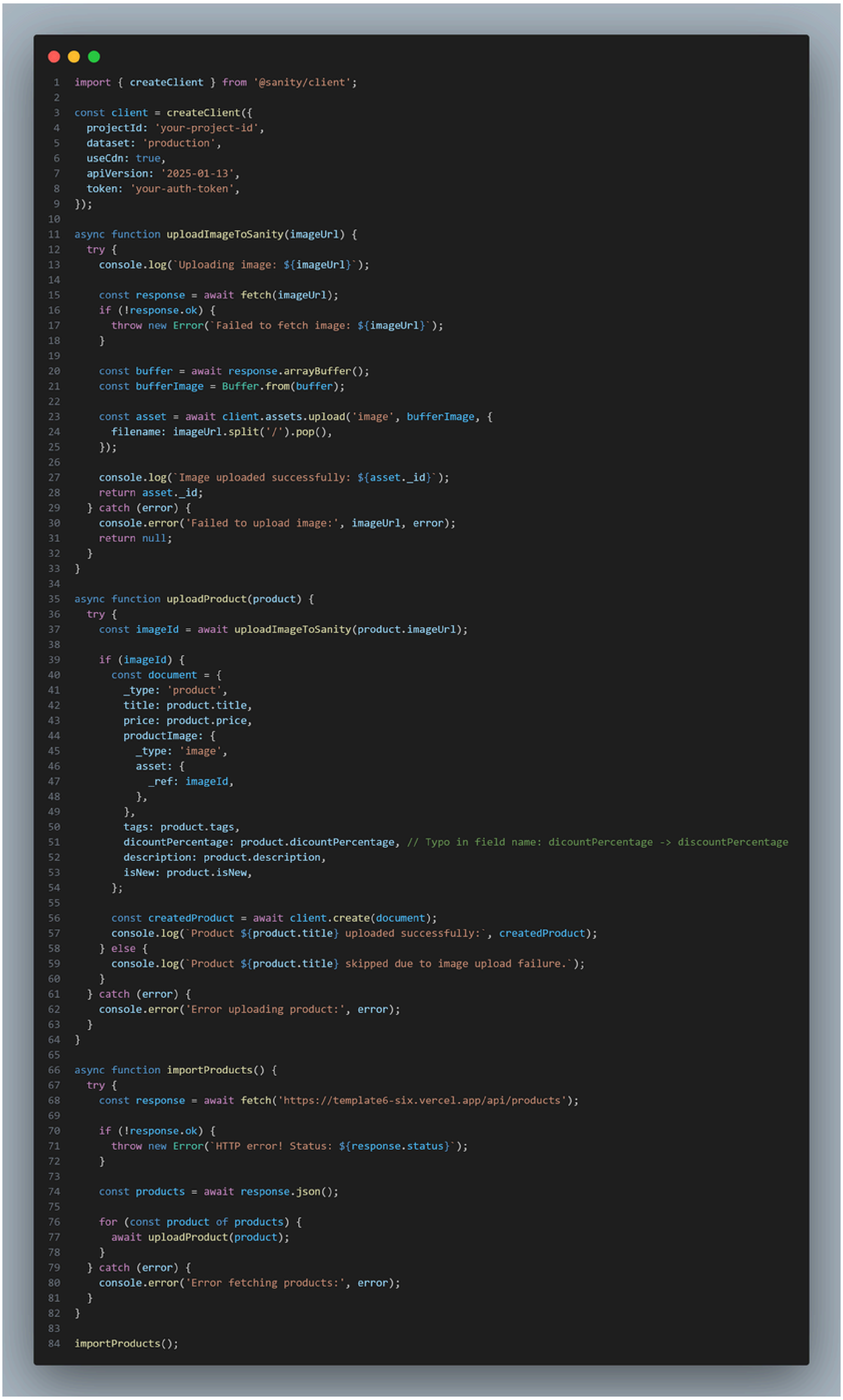
Data Migration Plan

1. **Fetch Data:** Used API to retrieve data.
2. **Transformation:** Converted data to match new schema.
3. **Import Using Templates:** Utilized templates for seamless data import.
4. **Validation:** Implemented validation checks to ensure data integrity.

Fetch method is used

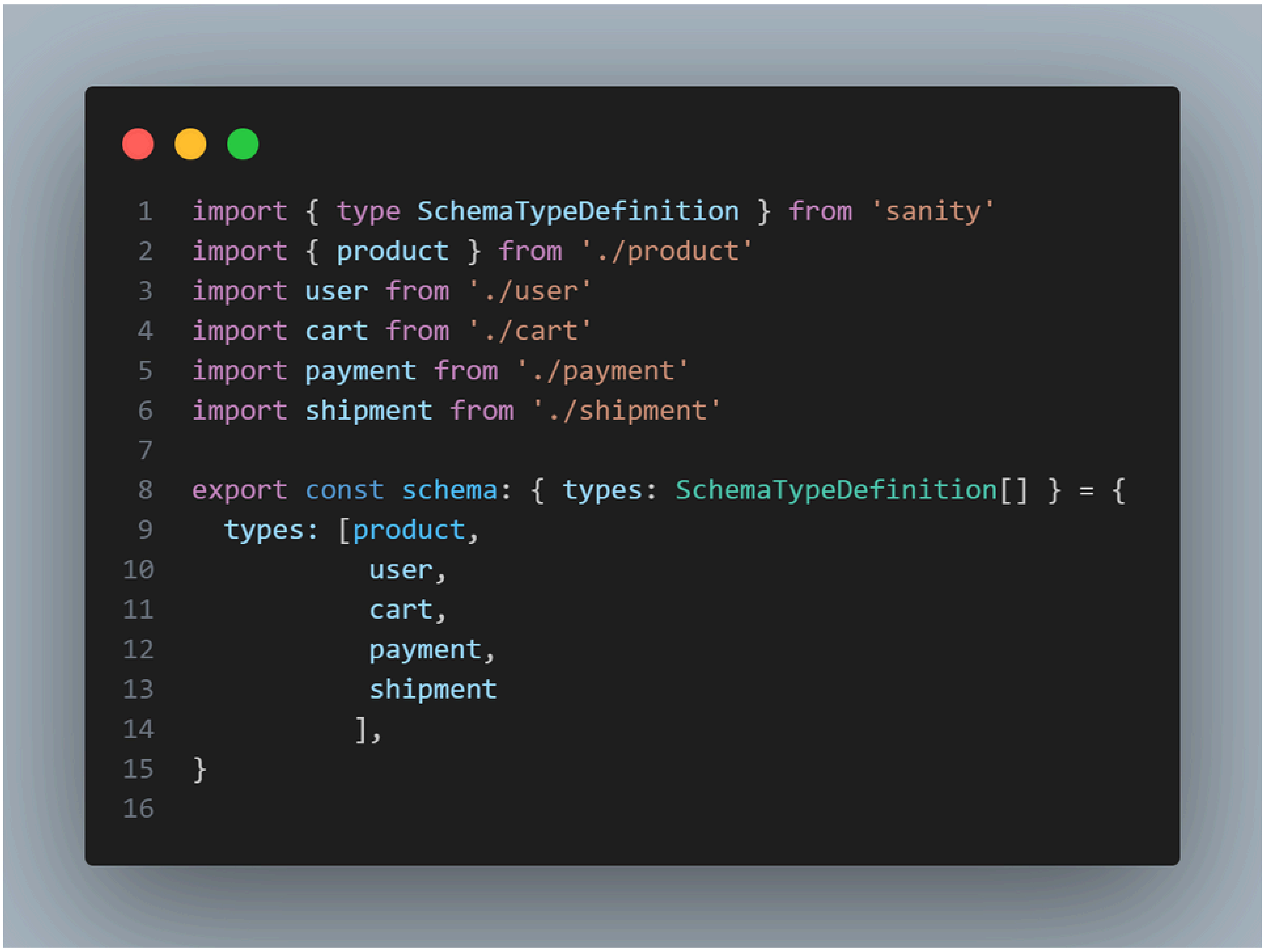


product/page.tsx



```javascript
import { createClient } from '@sanity/client';

const client = createClient({
  projectId: 'your-project-id',
  dataset: 'production',
  useCdn: true,
  apiVersion: '2025-01-13',
  token: 'your-auth-token',
});

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(),
    });

    console.log(`Image uploaded successfully: ${asset._id}`);
    return asset._id;
  } catch (error) {
    console.error('Failed to upload image:', imageUrl, error);
    return null;
  }
}

async function uploadProduct(product) {
  try {
    const imageId = await uploadImageToSanity(product.imageUrl);

    if (imageId) {
      const document = {
        _type: 'product',
        title: product.title,
        price: product.price,
        productImage: {
          _type: 'image',
          asset: {
            _ref: imageId,
          },
        },
        tags: product.tags,
        dicountPercentage: product.dicountPercentage, // Typo in field name: dicountPercentage -> discountPercentage
        description: product.description,
        isNew: product.isNew,
      };

      const createdProduct = await client.create(document);
      console.log(`Product ${product.title} uploaded successfully:`, createdProduct);
    } else {
      console.log(`Product ${product.title} skipped due to image upload failure.`);
    }
  } catch (error) {
    console.error('Error uploading product:', error);
  }
}

async function importProducts() {
  try {
    const response = await fetch('https://template6-six.vercel.app/api/products');

    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }

    const products = await response.json();

    for (const product of products) {
      await uploadProduct(product);
    }
  } catch (error) {
    console.error('Error fetching products:', error);
  }
}

importProducts();
```

Index.ts



```
import { type SchemaTypeDefinition } from 'sanity'
import { product } from './product'
import user from './user'
import cart from './cart'
import payment from './payment'
import shipment from './shipment'

export const schema: { types: SchemaTypeDefinition[] } = {
  types: [product,
           user,
           cart,
           payment,
           shipment
          ],
}

```

Error Handling During Migration

- **Mechanisms:** Logging and alerting for errors.
- **Recovery:** Automated retries for failed migrations.

Next.js Integration

API Integration in Next.js

- **Utility Functions:** Created utility functions in src/script/api.js.
- **Component Rendering:** Rendered data in components such as ourProduct.tsx, shop.tsx, and relatedProduct.tsx.
- **Testing:** Utilized Thunder Client and console.log for testing API integration.

Error Handling

Error Handling Mechanisms

- **Error Components:** Implemented error handling in /app/layout.tsx, /app/error.tsx, and /dashboard/page.tsx.
- **Exception Handling:** Used try-catch blocks to handle exceptions and ensure smooth program execution.

Next.js Integration

Utility Functions

- **Location:** src/script/api.js
- **Functions:** fetchData(), postData()

Component Rendering

- **Components:**
  - ourProduct.tsx: Displays product details.
  - shop.tsx: Handles shopping logic.
  - relatedProduct.tsx: Shows related products.

Testing

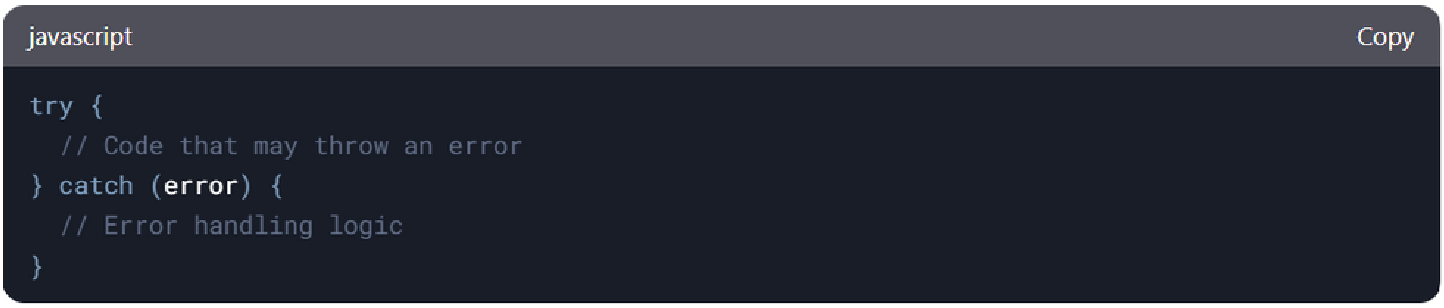
- **Tools:** Thunder Client, console.log for debugging.

Error Components

- **Files:**
  - /app/layout.tsx
  - /app/error.tsx
  - /dashboard/page.tsx

Exception Handling

- **Code Example:**


javascript Copy

```javascript
try {
  // Code that may throw an error
} catch (error) {
  // Error handling logic
}
```

Execution Strategy

- **Continuation:** Ensured program continues after catching exceptions.

Day 3 Checklist

- **API Understanding:** Completed deep dive into API documentation.
- **Schema Validation:** Ensured schemas are validated and adjusted.
- **Data Migration:** Successfully migrated data using the provided API.
- **API Integration in Next.js:** Integrated APIs into Next.js components.
- **Submission Preparation:** Prepared the project for submission.

Acknowledgments

- **Team Members:** Special thanks to [Names] for their contributions.
- **Tools:** Postman, Thunder Client, and Next.js framework.

References

- **API Documentation:** [Link to API docs]
- **Schema References:** [Links to schema files]
- **Hackathon Guidelines:** [Link to hackathon guidelines]