Marketplace Technical Foundation - At-Door

Table of Contents

1. System Architecture Overview

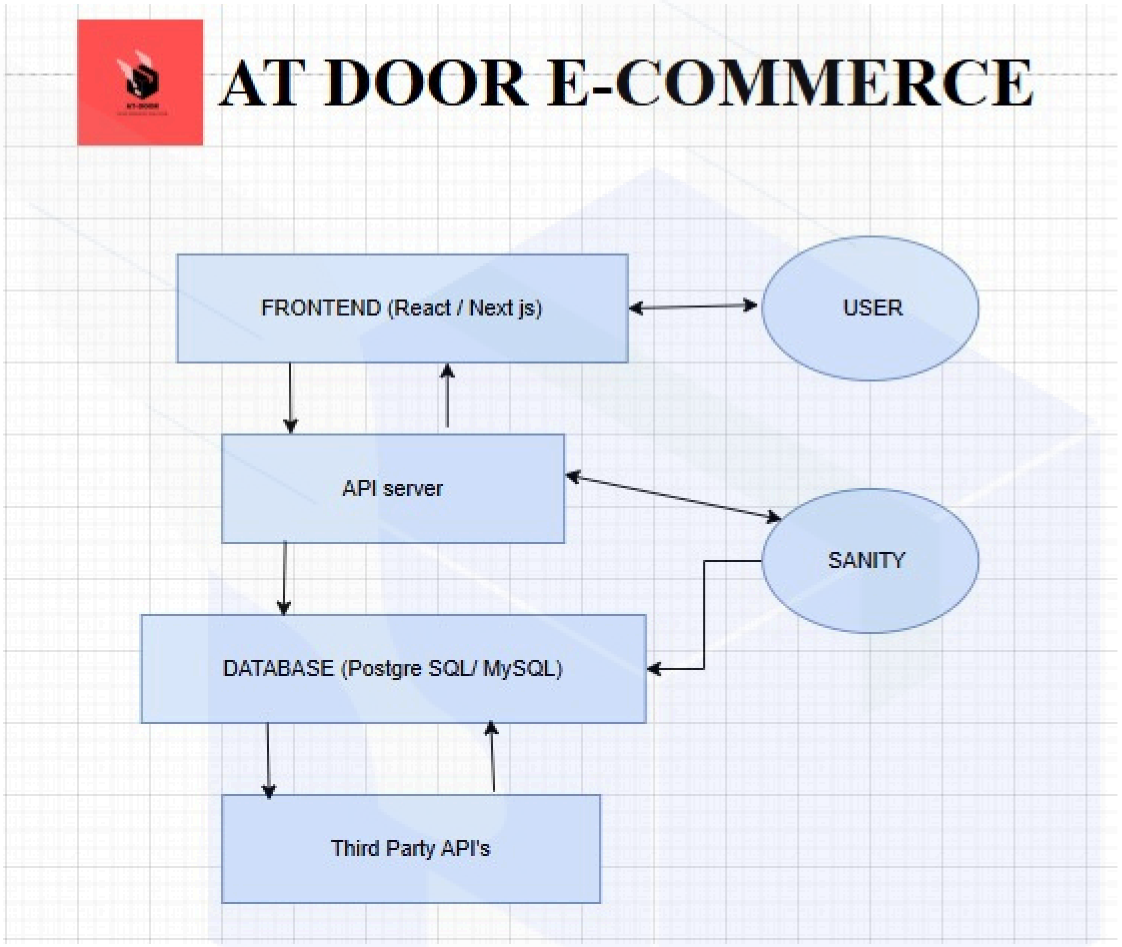
The **At-Door** platform utilizes a modern tech stack for seamless user interaction, robust content management, and efficient third-party integrations. Below is a high-level overview of the system architecture:

Components:

- **Frontend (User Interface)**: React.js with TypeScript
  - o Provides the user-facing interface for browsing, ordering, and user account management.
- **Backend (API Server)**: Java (Spring Boot)
  - o Manages user authentication, business logic, and interaction with the database.

- **Sanity CMS**: Headless CMS for managing dynamic content such as product details, blog posts, and promotional content.
- **Third-party APIs**: Payment gateways (e.g., Stripe), shipping APIs (e.g., Shippo), email services (e.g., SendGrid).

System Diagram:



Explanation:

1. **Frontend**: The React-based frontend provides a seamless shopping experience, fetching content dynamically from Sanity CMS and interacting with the backend API for data storage and management.
2. **Backend API**: The Java-based backend serves as the central server handling requests from the frontend, querying data from the database, and interacting with third-party APIs for payments and shipping.

3. **Sanity CMS**: Sanity allows content editors to easily manage products, promotions, and dynamic content. It interacts with the backend to dynamically serve content.
4. **Database**: A relational database (e.g., PostgreSQL or MySQL) stores product data, user information, orders, and more.
5. **Third-party APIs**: Services like Stripe for payments and Shippo for shipping management.

2. Key Workflows

User Registration:

1. **Frontend**:
   a. User accesses the registration page and inputs personal details.
   b. Form is validated and submitted to the API server.
2. **Backend (API)**:
   a. The API validates the input data and checks if the email is unique.
   b. User data is hashed and saved in the database.
3. **Sanity CMS**:
   a. Not involved in this workflow directly, but any content related to user guidance (e.g., registration tips) can be stored here.
4. **Frontend**:
   a. A confirmation email is sent (via a third-party service like SendGrid).
   b. The user is redirected to the homepage or dashboard once registered.

Product Browsing:

1. **Frontend**:
   a. User browses categories, views product listings, and clicks on a product.
2. **Backend (API)**:
   a. The API queries the database for products and their details.
   b. The API also fetches promotional content stored in Sanity CMS.
3. **Sanity CMS**:
   a. Provides dynamic content such as promotions, banners, and special offers to be displayed alongside products.
4. **Frontend**:
   a. Products and promotions are displayed dynamically using React components.

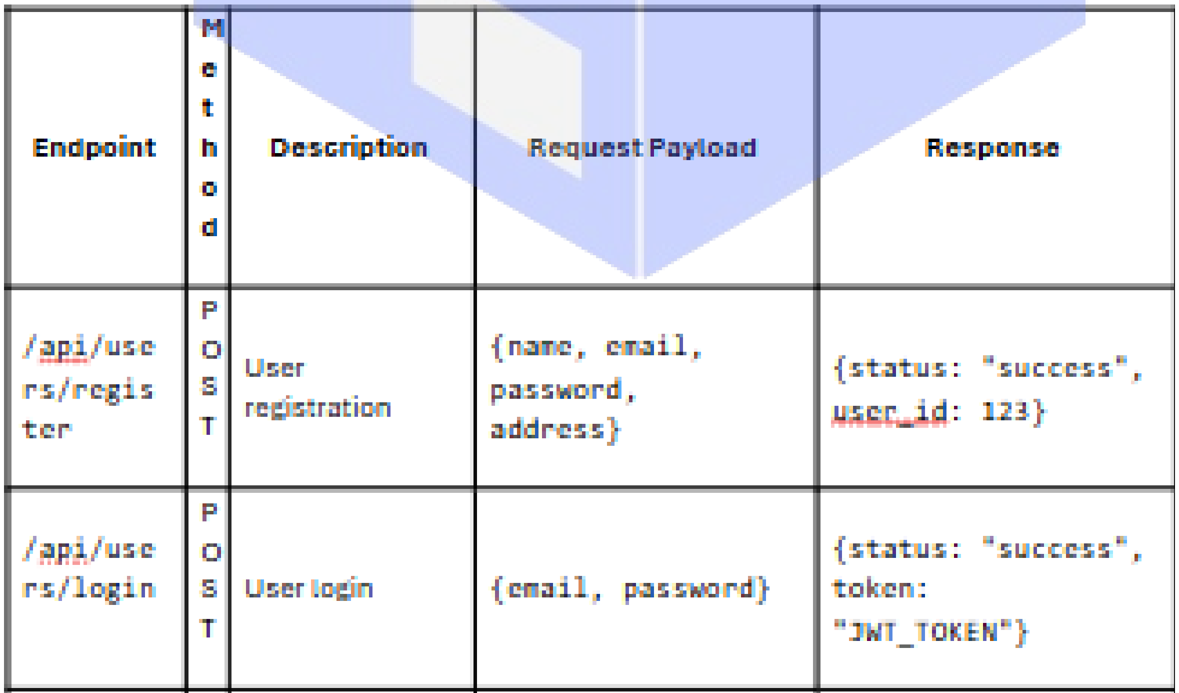
Order Placement:

1. **Frontend**:
    a. User adds products to the cart and proceeds to checkout.
2. **Backend (API)**:
    a. API receives the order details and checks product availability.
    b. The API integrates with the payment gateway (e.g., Stripe) for payment processing.
    c. Once payment is successful, an order record is created in the database.
3. **Sanity CMS**:
    a. Not involved directly in the order placement, but can be used to display order status or related information (e.g., shipping details).
4. **Third-party APIs**:
    a. The API communicates with the shipping API (e.g., Shippo) to calculate shipping costs and generate labels.
    b. The user receives an email notification with order details and tracking info.

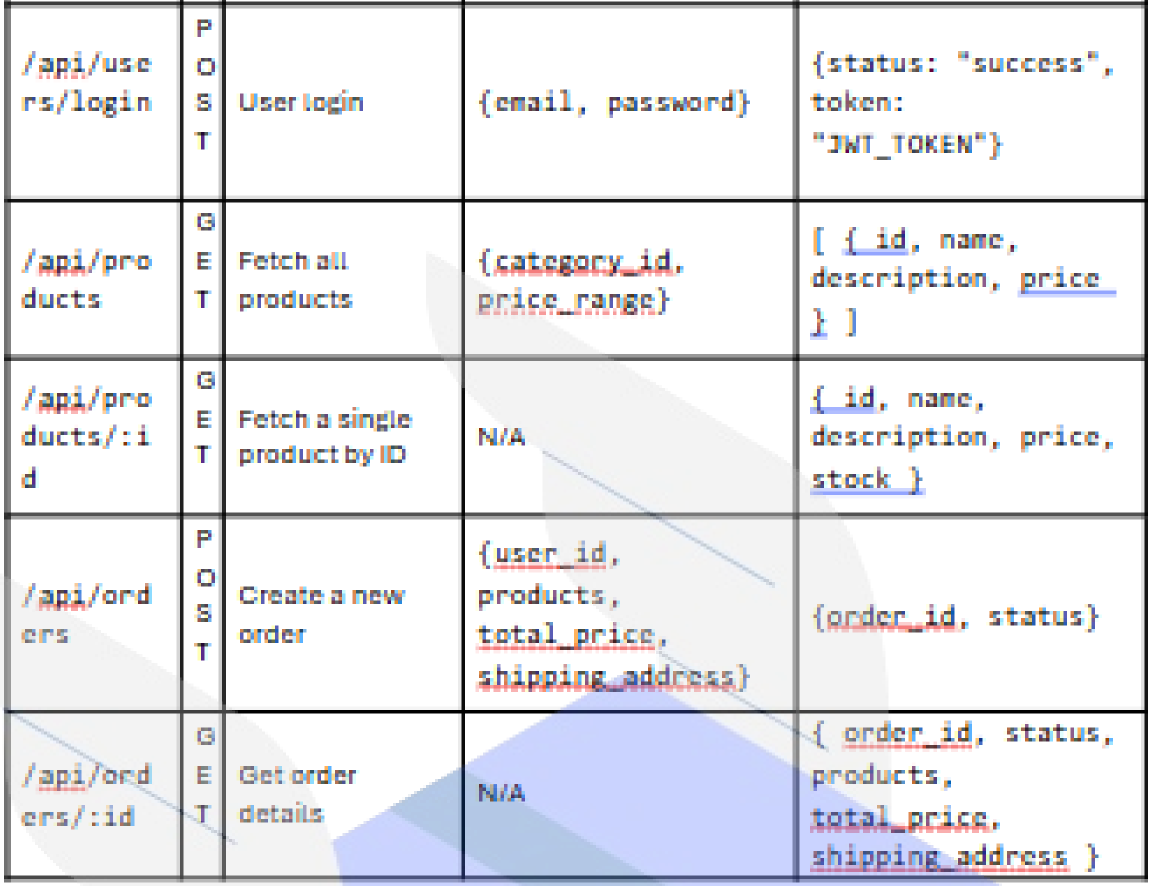
3. API Requirements

The backend API will expose several endpoints to interact with the frontend and third-party services. **The following are the simplest method for adding API requests in the code, however some requests in the original code (Template 6) may vary.**

Endpoints:



Endpoint	Method	Description	Request Payload	Response
/api/users/register	POST	User registration	{name, email, password, address}	{status: "success", user_id: 123}
/api/users/login	POST	User login	{email, password}	{status: "success", token: "JWT_TOKEN"}



/api/users/login	POST	User login	{email, password}	{status: "success", token: "JWT_TOKEN"}
/api/products	GET	Fetch all products	{category_id, price_range}	[{ _id, name, description, price }]
/api/products/:id	GET	Fetch a single product by ID	N/A	{ _id, name, description, price, stock }
/api/orders	POST	Create a new order	{user_id, products, total_price, shipping_address}	{order_id, status}
/api/orders/:id	GET	Get order details	N/A	{ order_id, status, products, total_price, shipping_address }

Example - Register Use(POST /api/users/register)

Request:

```
 {

   "name": "John Doe",
  "email": "john.doe@example.com",
"password": "securepassword",
"address": "123 Main St, Anytown"
}
```

Response:

```
json Copy
{

  "status": "success",
  "user_id": 123
}
```

4. Sanity Schema Design :

Below is the data schema that outlines how data is structured in **Sanity** and how it's consumed in your Next.js app:

1. Product Schema (sanity/schemaTypes/product.ts)

This schema defines how product data is structured in Sanity and corresponds to how products are represented in your Next.js frontend.

```
// sanity/schemaTypes/product.ts
export default {

 name: 'product',
 title: 'Product',
 type: 'document',
 fields: [
  {
   name: 'name',
   title: 'Name',
   type: 'string',
   validation: Rule => Rule.required().min(1).max(100),
  },
  {
   name: 'description',
   title: 'Description',
   type: 'text',
   validation: Rule => Rule.required().min(10),
  },
  {
   name: 'price',
   title: 'Price',
   type: 'number',
   validation: Rule => Rule.required().min(0),
  },
  {
   name: 'category',
   title: 'Category',
   type: 'reference',
   to: [{ type: 'category' }],
  },
  {
   name: 'image',
   title: 'Image',
   type: 'image',
   options: {
    hotspot: true,
   },
  },
  {
   name: 'stockQuantity',
   title: 'Stock Quantity',
   type: 'number',
   validation: Rule => Rule.required().min(0),
  },
 ],
}
```

2. Category Schema (sanity/schemaTypes/category.ts)

This schema defines product categories.

```
// sanity/schemaTypes/category.ts
export default {
 name: 'category',
 title: 'Category',
 type: 'document',
 fields: [
  {
   name: 'name',
   title: 'Category Name',
   type: 'string',
    validation: Rule => Rule.required().min(1).max(50),
  },
  {
    name: 'description',
    title: 'Description',
    type: 'text',
  },
 ],
}
```

3. Order Schema (sanity/schemaTypes/order.ts)

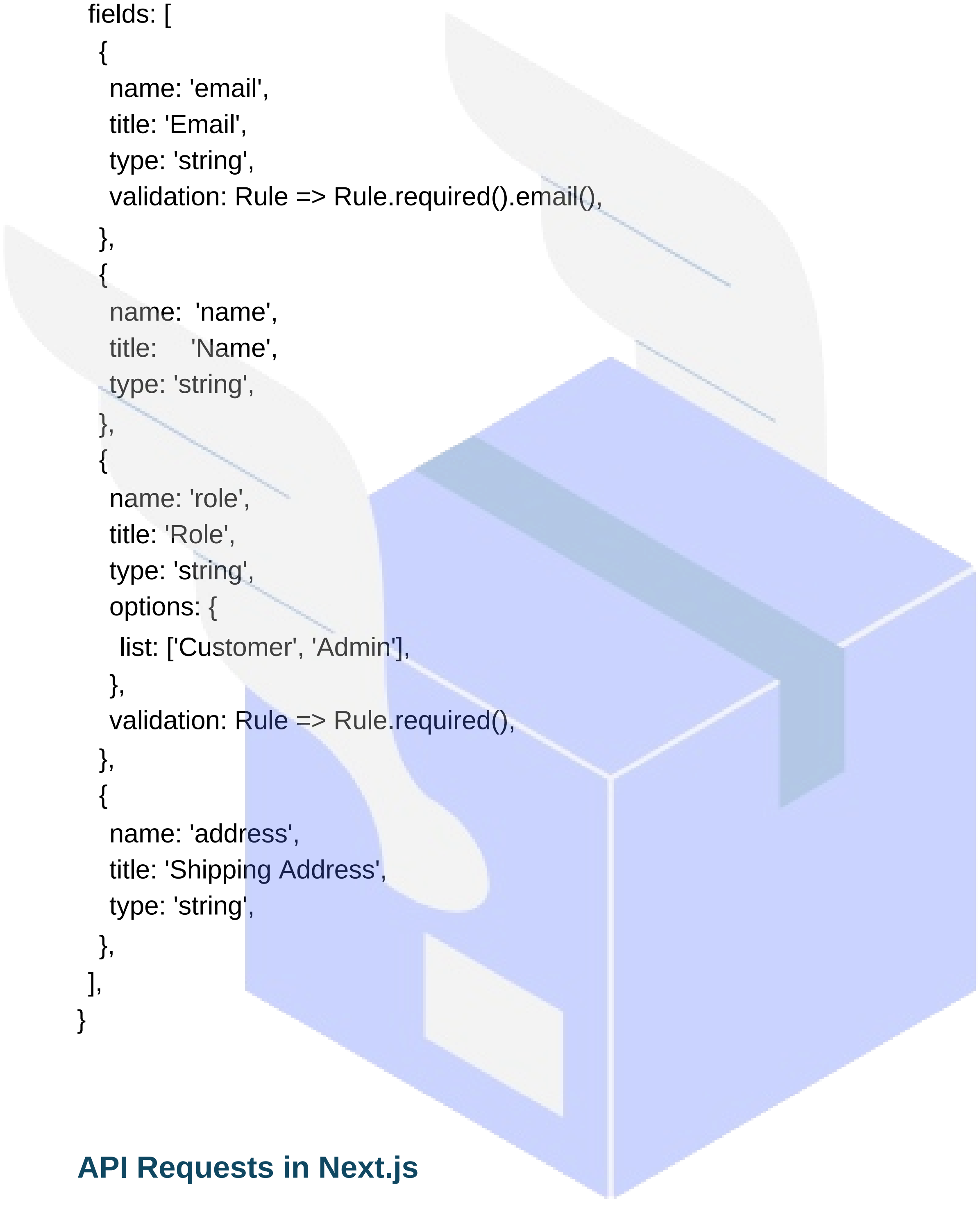
This schema defines how order information will be stored in Sanity. Orders are typically managed outside Sanity (in a backend database), but if you need to track order data here, you could use this schema.

```
// sanity/schemaTypes/order.ts
export default {
name: 'order',
title: 'Order',
type: 'document',
fields: [
  {
    name: 'user',
    title: 'User',
    type: 'reference',
    to: [{ type: 'user' }],
  },
  {
    name: 'products',
    title: 'Products',
    type: 'array',
    of: [{ type: 'reference', to: [{ type: 'product' }] }],
  },
  {
    name: 'totalPrice',
    title: 'Total Price',
    type: 'number',
    validation: Rule => Rule.required().min(0),
  },
  {
    name: 'status',
    title: 'Status',
    type: 'string',
    options: {
     list: ['Pending', 'Shipped', 'Delivered'],
    },
    validation: Rule => Rule.required(),
  },
  {
    name: 'shippingAddress',
    title: 'Shipping Address',
    type: 'string',
  },
  {
    name: 'paymentStatus',
    title: 'Payment Status',
    type: 'string',
    options: {
     list: ['Pending', 'Paid', 'Failed'],
    },
    validation: Rule => Rule.required(),
  },
 ],
}
```

4. User Schema (sanity/schemaTypes/user.ts)

Define user-related data (assuming you track users in Sanity).

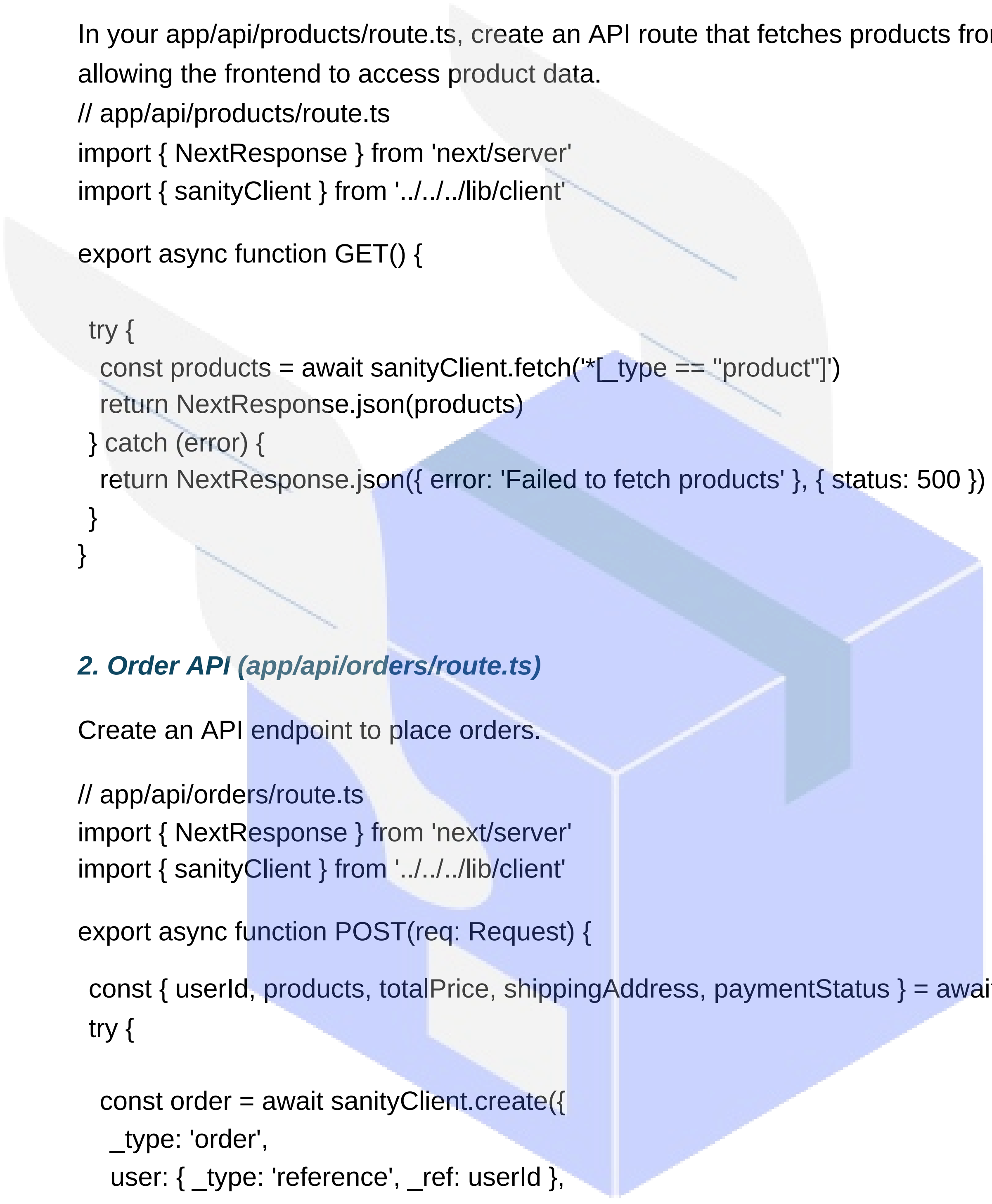
```
// sanity/schemaTypes/user.ts
export default {
name: 'user',
title: 'User',
```



```
 type: 'document',
 fields: [
  {
   name: 'email',
   title: 'Email',
   type: 'string',
   validation: Rule => Rule.required().email(),
  },
  {
   name:  'name',
   title:     'Name',
   type: 'string',
  },
  {
   name: 'role',
   title: 'Role',
   type: 'string',
   options: {
    list: ['Customer', 'Admin'],
   },
   validation: Rule => Rule.required(),
  },
  {
   name: 'address',
   title: 'Shipping Address',
   type: 'string',
  },
 ],
}
```

API Requests in Next.js

Below are the API routes in Next.js (using the app/api/products/route.ts file) for handling CRUD operations for products, orders, and users.



1. Product API (app/api/products/route.ts)

In your app/api/products/route.ts, create an API route that fetches products from Sanity, allowing the frontend to access product data.

```
// app/api/products/route.ts
import { NextResponse } from 'next/server'
import { sanityClient } from '../../../lib/client'

export async function GET() {

  try {
    const products = await sanityClient.fetch('*[_type == "product"]')
    return NextResponse.json(products)
  } catch (error) {
    return NextResponse.json({ error: 'Failed to fetch products' }, { status: 500 })
  }
}
```

2. Order API (app/api/orders/route.ts)

Create an API endpoint to place orders.

```
// app/api/orders/route.ts
import { NextResponse } from 'next/server'
import { sanityClient } from '../../../lib/client'

export async function POST(req: Request) {

  const { userId, products, totalPrice, shippingAddress, paymentStatus } = await req.json()
  try {

    const order = await sanityClient.create({
      _type: 'order',
      user: { _type: 'reference', _ref: userId },
      products: products.map((productId: string) => ({
        _type: 'reference',
        _ref: productId,
      })),
      totalPrice,
      status: 'Pending',
      shippingAddress,
      paymentStatus,
    })
    return NextResponse.json(order, { status: 201 })

  } catch (error) {
    return NextResponse.json({ error: 'Failed to create order' }, { status: 500 })
  }
}
```

3. User API (app/api/users/route.ts)

Handle user authentication and management (e.g., login, registration).

```
// app/api/users/route.ts
import { NextResponse } from 'next/server'
import { sanityClient } from '../../../lib/client'

export async function POST(req: Request) {

 const { email, name, role, address } = await req.json()
 try {

  const user = await sanityClient.create({
    _type: 'user',
    email,
    name,
    role,
    address,
  })
  return NextResponse.json(user, { status: 201 })

 } catch (error) {
  return NextResponse.json({ error: 'Failed to create user' }, { status: 500 })
 }
}
```

Sanity Client Setup (lib/client.ts)

Set up a Sanity client to interact with your Sanity CMS.

```
// lib/client.ts
import sanityClient from '@sanity/client'
export const client = sanityClient({

  projectId: 'your_project_id', // Replace with your Sanity project ID
  dataset: 'production', // The dataset to use
  apiVersion: '2023-01-01', // Date of the API version
  token: 'your_token', // Optional: for private access
  useCdn: true, // Use the CDN for fast responses
})
```

Sanity Data Fetching

You can fetch data from Sanity CMS by using the client in API routes or components, like in the app/api/products/route.ts above.

5. Collaboration Notes

The development of the **At-Door** marketplace involved close collaboration between frontend developers, backend developers, and content managers. Below are the key insights:

- **Frontend & Backend Collaboration**: The integration between the frontend (React) and backend (Java Spring Boot) involved multiple API iterations. The frontend team provided regular feedback on API responses and required endpoints, which were incorporated to ensure smooth user interactions.
- **Sanity CMS**: The content management schema was developed in parallel with the product and order schemas. The backend was designed to pull dynamic content from Sanity, while the frontend integrated this data seamlessly for a personalized user experience.

- **Challenges Faced**:
  - **API Design**: One of the major challenges was ensuring that the API endpoints were designed to handle both product data and user-related data efficiently.
  - **Third-party Integrations**: Integrating Stripe and Shippo APIs required careful handling of asynchronous requests to ensure that payments and shipping information were processed smoothly.
- **Feedback**:
  - The API documentation was continuously improved based on feedback from both frontend and backend teams to ensure that endpoints were consistent and easy to use.

Conclusion

This setup provides a comprehensive **data schema design** for your **At-Door** marketplace, including API routes for handling product data, order management, and user data. The **Sanity CMS** serves as the content management layer, with **Next.js API routes** handling CRUD operations and interactions between the frontend and backend.

- **Product Data** is fetched from Sanity via the /api/products API.
- **Order Creation** is done through the /api/orders API.
- **User Management** is handled through the /api/users API.