



**College Code : 9509**

**College Name : Holy Cross Engineering College**

**Department : Computer Science Engineering**

**Student NM id : E11EF02A38085BF808C481335F5D08E1**

**Roll No : 950923104027**

**Date : 15.09.2025**

**Project Name : IBM-FE-Product Catalog with Filters**

**Completed the project named as**

**Sumbitted By,**

**Mareeswaran C**

**9150430858**

## Introduction

This document outlines the technical blueprint for the product catalog with filters project. Building on the requirements defined in Phase 1, this phase details the core architectural decisions and design patterns we will use to build a scalable, high-performance, and maintainable application. It covers our chosen **technology stack**, the proposed **UI and API structures**, our approach to **handling data**, and visual diagrams to illustrate the system's components and user flows. The primary goal of this design is to create a robust foundation that not only meets the immediate MVP requirements but also supports future growth and feature enhancements. 🚀

## Tech Stack Selection

For this project, we'll use a modern, scalable, and widely-supported tech stack known for its performance and developer efficiency. The choice is the **MERN stack**, with some specific library selections.

- **Frontend: React.js**
  - **Why?** React's component-based architecture is perfect for building a modular UI like our product catalog. Its virtual DOM ensures fast rendering and updates, which is crucial when filters are applied and the product list changes frequently. The vast ecosystem (state management, routing) and large community provide excellent support.
- **Backend: Node.js with Express.js**

- **Why?** Node.js is a non-blocking, event-driven runtime, making it highly efficient for handling concurrent API requests from many users. Express.js is a minimal and flexible framework that simplifies the creation of a robust REST API for serving product data. Using JavaScript on both the front and back end streamlines development.
- **Database: MongoDB**
  - **Why?** As a NoSQL document database, MongoDB is extremely flexible. Our product schema can evolve easily without complex migrations. It's well-suited for storing product data, which can have varied and nested attributes. Its querying capabilities are powerful enough to handle the complex filtering and sorting logic required for the catalog.
- **Deployment / Hosting: Vercel (for Frontend) and MongoDB Atlas (for Database)**
  - **Why?** Vercel offers a seamless, zero-configuration deployment experience for React applications with built-in CI/CD. MongoDB Atlas is a fully managed cloud database service that handles scaling, backups, and security, allowing our team to focus on development rather than database administration.

**UI Structure / API Schema Design**

**UI Component Structure (React)**

The frontend will be broken down into reusable, self-contained components:

- **CatalogPage.js:** The main container component that orchestrates all other components. It holds the primary state for filters and the product list.
- **SearchBar.js:** A controlled component for text-based search input.
- **FilterSidebar.js:** A container for all filtering options.
  - **CategoryFilter.js:** Renders a list of categories.
  - **PriceFilter.js:** Renders a price range slider or input fields.
  - **BrandFilter.js:** Renders a list of brand checkboxes.
- **ProductGrid.js:** Maps over the product data and renders a ProductCard for each item. It also contains the SortDropdown.js.
- **ProductCard.js:** Displays a single product's image, name, and price.
- **Pagination.js:** Handles page navigation for the product list.

## API Schema Design

The API will expose clear, RESTful endpoints.

### 1. Product Model (Product Schema in MongoDB)

JSON

```
{  
  "_id": "ObjectId",
```

```
"name": "String",  
"sku": "String",  
"description": "String",  
"price": "Number",  
"category": {  
  "id": "ObjectId",  
  "name": "String"  
},  
"brand": {  
  "id": "ObjectId",  
  "name": "String"  
},  
"imageUrl": "String",  
"stock": "Number",  
"attributes": [  
  { "key": "color", "value": "Blue" },  
  { "key": "size", "value": "M" }  
],  
"createdAt": "Date",  
"popularityScore": "Number"  
}
```

## 2. GET /api/products - Fetch Filtered Products

- **Response Schema:**

JSON

```
{
  "pagination": {
    "currentPage": 1,
    "totalPages": 10,
    "totalProducts": 100
  },
  "products": [
    {
      "_id": "63e8c4e5a1b2c3d4e5f6a7b8",
      "name": "Classic Cotton T-Shirt",
      "price": 25.99,
      "brand": { "name": "Brand A" },
      "imageUrl": "https://example.com/image1.jpg"
    }
  ]
}
```

## 3. GET /api/filters - Fetch Available Filter Options

- **Response Schema:**

## JSON

```
{  
  "categories": [  
    { "id": "cat1", "name": "T-Shirts" },  
    { "id": "cat2", "name": "Jeans" }  
  ],  
  "brands": [  
    { "id": "brand1", "name": "Brand A" },  
    { "id": "brand2", "name": "Brand B" }  
  ],  
  "priceRange": {  
    "min": 10,  
    "max": 500  
  }  
}
```

## Data Handling Approach

- **Frontend State Management:** We will use **Zustand**, a simple and powerful state management library for React. A central store will manage the global state, including the current list of products, active filters, pagination state, and loading/error states. When a user applies a filter, the component updates the

Zustand store, which automatically triggers a new API call to fetch the updated product list.

- **Backend Data Processing:** The Node.js/Express server will handle incoming requests to `/api/products`.
  1. **Validation:** It will first validate and sanitize all query parameters (category, minPrice, sortBy, etc.).
  2. **Query Construction:** It will dynamically build a MongoDB query object based on the validated parameters.
  3. **Database Indexing:** To ensure fast query performance, the MongoDB products collection will have **indexes** on key filterable fields: category.id, brand.id, price, and popularityScore. This is critical for performance at scale.
  4. **Serialization:** The data retrieved from MongoDB will be formatted into the defined API schema before being sent back as a JSON response.

## Component / Module Diagram

This diagram shows the high-level architecture and the flow of information between the main components of the system.

### Client (Browser)

- **React Application:**
  - UI Components (FilterSidebar, ProductGrid, etc.)
  - State Management (Zustand Store)
  - API Client (e.g., Axios)



- *Sends HTTP requests (e.g., GET /api/products?brand=BrandA)*

↑ ↓ (HTTP/JSON)

## Server (Node.js/Vercel Serverless Function)




- **Express.js API:**
  - Routes (/products, /filters)
  - Controllers (Handle request logic)
  - Services (Business logic, query building)
    - *Constructs and sends database queries*

↑ ↓ (Database Connection)

## Database (MongoDB Atlas)

- **MongoDB:**
  - Collections (products, categories, brands)
  - Indexes (On price, brand.id, etc.)
    - *Executes queries and returns documents*

## Basic Flow Diagram

This flowchart outlines the sequence of events when a user applies a filter.   

1. **User Action:** User clicks the "Brand A" checkbox in the FilterSidebar.

2. **Frontend State Update:** The onClick event handler calls a function to update the central Zustand store. The active filters state now includes "brand": "Brand A".
3. **API Request Triggered:** A useEffect hook in the CatalogPage component, subscribed to the filter state, detects the change and triggers a new API request: GET /api/products?brand=BrandA. A loading spinner is shown in the UI.
4. **Backend Processing:** The Express server receives the request and validates the brand query parameter.
5. **Database Query:** The server constructs a MongoDB find query: db.products.find({ "brand.name": "Brand A" }).
6. **Data Retrieval:** MongoDB uses its index on the brand.name field to efficiently find all matching products and returns the data to the server.
7. **Backend Response:** The server formats the data into the JSON schema (with pagination info) and sends it back to the client with a 200 OK status.
8. **UI Update:** The frontend API client receives the JSON response, updates the Zustand store with the new products, and sets the loading state to false. React automatically re-renders the ProductGrid component to display only products from "Brand A". ✨

## Conclusion

The solution design detailed in this document provides a clear and comprehensive technical roadmap for the development team. By selecting the **MERN stack**, we leverage a modern, cohesive, and efficient set of technologies. The component-based UI architecture, well-defined API schemas, and a clear data handling strategy will enable us to build the application in a structured and scalable manner. The accompanying diagrams offer a high-level view of the system's architecture, ensuring all team members have a shared understanding of how the components interact. This architectural plan is robust and provides a solid foundation as we move into the next phase