College code:9607

College Name: Immanuel arasar jj college of engineering

Department:BE.CSE

Student NM_id:aut2396070020

Roll number :960723104018

Date :26.09.2025

Completed the project named as Phase_M.Mohamed Nabi,R.Karpagam,M.Santhiya,T.varshini,M.Paneer Selvam.

Technology Project Name:Product catalog With Filters.

Submitted by,

Name:R.Karpagam
Mobile no:9894303802

Product catalog with filters

Phase 4

Enchancement &Deployment

Content:

1. **Additional features**
2. **UI/UX improvements**
3. **API Enchancements**
4. **Performance&Security Checks**
5. **Testing of Enhancement**

6.Deployment (Netlify,vercel or Cloud platform)

1. Additional Features:

I. More Complex Filtering Choices:

1.Users can choose more than one value (e.g., numerous colors, brands) with multi-select filters.

2.Range sliders ? For weight, rating, price, or discount.

3.Large filter lists can benefit from the ability to search within filters (e.g., searching within "brands").

4.Dependent filters ? Only when a primary filter is selected will sub-filters be displayed (e.g., choose "Laptop" ? then reveal "RAM size").

II. organizing and Sorting:

1.Sort by rating, discount, price (low ? high / high ? low), popularity, and newest.

2.Products that are pinned or featured are always at the top.

3.For easier navigation, group by brand or category.

III. Improvements to User Experience:

1.No need to refresh; live filter updates are available.
Click the "Clear all filters" option.

2. To display the selected filters with the delete option, place filter chips or tags at the top.

3.Filter portions that collapse for a tidy user interface.

4.Save filter settings for people that are currently logged in.

IV. Features for Personalization:

1.suggested filters according on the user's past web activity.

2.Quick access to recently used filters.

3.Features That Are Mobile-Friendly:

, 4.mobile screen filter panel that slides in.

5.For convenience, a sticky apply button is located at the bottom.

V..Features specific to the product:

1.only toggle that is in stock.

2.Cash on delivery and free shipping filters.

3.discounts and offers filter.

4.Filter for new arrivals and limited stocks.

2.UI/UX improvements:

I.UI Enhancement:

Layout of a Catalog :

1. Products are displayed in a grid format, with two to four ite ms.per row, depending on screen size.

2. .Large, excellent product photos with easy view and hover z oom.

Uniformity in the card's image, title, rating, price, and call to action button.

Design of Filters:

Desktop:: Collapsible filter sections in a sticky left sidebar.

Mobile: Filters in a slide-in drawer or bottom sheet.

For faster selection, use toggles, sliders, and checkbox rather than drop down menus.

Visual Improvements :

1.Above the catalog, use tags or pills to draw attention to the app lied filters (Color: Red ×).

To improve feedback, add hover states to buttons and cards.

Visually display whether a product is in-stock or out-of-stock.

II.Enhancements to UX:

Experience Filtering :

Instant filtering (results that automatically update) or easily acces sible Apply/Reset buttons.

filters with numerous selections (e.g., select multiple colors or sizes).

Use the "Clear All" option to rapidly reset the filters.

Searching and Sorting :

1. Sorting options include Best Rated, New Arrivals, and Price (low to high). intelligent search with spelling tolerance and auto-complete.

Performance & Feedback :

When results load, use shimmer effects or skeleton loader.

Show the quantity of items discovered ("23 items fit your filters").

Display the "No Results" condition along with recommendations, such as "Try eli minating certain criteria.

Personalization:

For returning users, keep in mind the last-used filters.

Emphasize suggested goods or popular categories.

Accessibility:

- Text and filter contrast is adequate.

Product cards and filters must to be accessible by keyboard.

For product photos, use alt text that describes the image.

III.Intuitive Placement & Selection:

- **Sticky Filters:** Keep filters visible on screen when scrolling to maintain find ability.
- **Smart Grouping:** Group related filters into logical categories (e.g., "Color," "Size," "Brand") to prevent overwhelming users.
- **Filter Summary:** Display active filters prominently at the top of the product grid so users know their current selections and can easily remove them.
- **<<"!nav>>Clear "Apply" and "Reset" Buttons:** Use distinct buttons for applying filter selections and clearing all active filters to guide the user's actions.
- **Visual Feedback:** Provide clear visual cues (e.g., checkmarks, highlighted text) when a filter is active to show the user its status.

IV.User Control & Customization:

- **Custom Filter Ordering:** Allow users to drag and drop to reorder filter options, enabling them to prioritize the filters most relevant to them.
- **Filter Navigation within Modals:** For a large number of filters, use a modal with internal navigation (like a tree-view or tabs) to prevent excessive scrolling.

V.Accessibility & Inclusivity:

- **Keyboard Navigation:** Ensure all filter controls are accessible and usable with a keyboard for users who cannot use a mouse.
- **Semantic HTML:** Use appropriate semantic HTML tags to provide context for screen readers and other assistive technologies.

Code Example (Conceptual HTML & JavaScript)

Code

```
<div class="product-catalog">
  <div class="filters-panel">
    <h3>Filters</h3>
    <!– Filter categories (e.g., Color, Price) –>
    <div class="filter-group">
      <h4>Color</h4>
      <ul>
```

```
            <li><input type="checkbox" class="filter-checkbox" data-filter-type="color"
data-value="red"> Red</li>
            <li><input type="checkbox" class="filter-checkbox" data-filter-type="color"
data-value="blue"> Blue</li>
          </ul>
        </div>
        <!– Other filter groups –>
      </div>

      <div class="product-grid">
        <div class="active-filters-summary">
          <!– Active filters displayed here –>
        </div>
        <div id="products-list">
          <!– Products will be rendered here –>
        </div>
        <div class="loading-indicator" style="display: none;">Loading...</div>
      </div>
    </div>
```

-

3.API Enchancement:

Core concepts and URL design

1. Basic filtering:

Use simple key-value pairs in the query string for exact matches on a single property.

Example: Find all products in the "electronics" category.

`GET /api/products?category=electronics`

2. Filtering with operators:

For more advanced filtering, use a consistent syntax to include operators like "greater than" (`gte`) or "less than" (`lt`).

Example: Find all products with a price between $10 and $100.

`GET /api/products?price[gte]=10&price[lte]=100`

Common operator syntax's:

- **Bracket syntax:** `price[gte]=10`

- **Suffix syntax:** `price_gt=50`

3. Filtering with multiple values:

Allow for multiple, comma-separated values to represent a logical `OR` condition.

Example: Find products in either the "books" or "electronics" category.
`GET /api/products?category=books,electronics`

4. Search queries:

For broader, text-based searches, a dedicated query parameter is best. This can be combined with other filter parameters.

Example: Search for "laptop" within the "electronics" category.
`GET /api/products?q=laptop&category=electronics`

Coding implementation example:

This example uses Python with the Flask framework and a list of dictionaries to represent the product catalog. The concepts are applicable to any language and framework.

1. API setup:

First, set up a basic API endpoint using a framework like Flask.

python

```
from flask import Flask, jsonify, request

app = Flask(__name__)

products = [
    {'id': 1, 'name': 'Wireless Mouse', 'category': 'electronics', 'price': 25.00, 'in_stock': True},
    {'id': 2, 'name': 'Mechanical Keyboard', 'category': 'electronics', 'price': 75.00, 'in_stock': True},
    {'id': 3, 'name': 'Laptop Stand', 'category': 'accessories', 'price': 30.00, 'in_stock': False},
```

```
    {'id': 4, 'name': 'Gaming Headset', 'category': 'electronics', 'price': 99.99, 'in_stock': True},
    {'id': 5, 'name': 'Notebook', 'category': 'office', 'price': 5.50, 'in_stock': True},
]
```

Use code with caution.

2. Implement filtering logic:

In your view function, access the URL query parameters using `request.args`. Create a function to apply filters dynamically.

```python
def apply_filters(products_list, filters):
    filtered_products = products_list

    # Filter by category (single or comma-separated values)
    if 'category' in filters:
        categories = filters['category'].split(',')
        filtered_products = [p for p in filtered_products if p['category'] in categories]

    # Filter by price range (gte and lte operators)
    if 'price[gte]' in filters:
        min_price = float(filters['price[gte]'])
        filtered_products = [p for p in filtered_products if p['price'] >= min_price]
    if 'price[lte]' in filters:
        max_price = float(filters['price[lte]'])
        filtered_products = [p for p in filtered_products if p['price'] <= max_price]
```

```
    # Filter by stock status
    if 'in_stock' in filters:
        is_in_stock = filters['in_stock'].lower() == 'true'
        filtered_products = [p for p in filtered_products if p['in_stock'] == is_in_stock]

    # Implement other filters here (e.g., search queries)

    return filtered_products
```

Use code with caution.

3. Create the API endpoint:

Integrate the filtering function into your API route.

```
python
@app.route('/api/products', methods=['GET'])def get_products():
    # Get all query parameters from the request
    filters = request.args

    # Apply the filters to the product catalog
    results = apply_filters(products, filters)

    return jsonify(results)
```

Use code with caution.

4. Run the application and test:

Start the server, then test the filters with a tool like Postman or a web browser.

Example 1: Filter by category

`GET HTTP://127.0.0.1:5000/API/products?category=electronics`

Example 2: Filter by multiple categories

`GET HTTP://127.0.0.1:5000/API/products?category=electronics,office`

Example 3: Filter by price range and stock status

`GET HTTP://127.0.0.1:5000/API/products?price[gte]=50&in_stock=true`

Advanced enhancements and best practices:

- **Pagination:** For large catalogs, always combine filtering with pagination to handle results efficiently. Add query parameters like `limit` and `offset`.
- **Performance:** Apply filters at the database query level to avoid retrieving unnecessary data from the database, which is much more efficient than filtering in application memory.
- **Validation:** Validate and sanitize all user inputs to prevent security vulnerabilities like SQL injection and to provide helpful error messages for invalid data.
- **Documentation:** Use clear, consistent naming conventions and provide comprehensive documentation so developers know which filters are available and how to use them effectively.
- **Caching:** Implement caching strategies for common filter combinations to speed up frequently requested data.
- **Logical operators:** Support more complex queries by allowing logical operators like `AND` (default for multiple parameters) and `OR`(often with comma- separated value)

4.Performance &security checks:

Performance Verification :

1. Database Effectiveness :

Make sure the fields used for filtering (such as category, pric e, brand, and rating) have the appropriate indexes.
Query Plans: To prevent entire table scans, examine queries (EXPLAIN in SQL).
Use effective pagination (either cursor-based or LIMIT / OFFSET).

1. Caching:

For popular filters (like "Best selling in Electronics"), use query caching.
If filters don't change frequently, use CDN caching for static content and API ans wers.

2. Search Engine Demoralization:

For quicker filtering and faceted search in huge catalogs, take into account Elast icsearch, Sol r, or Open Search.
Only the content that is viewable on the first page should be loaded using lazy lo ading (e.g., infinite scroll).

3. Backend & API :

1.Limiting rates to stop misuse.
2.If more than one filter is used, batch requests are made.
3.compression (Gzip, Brotli) to minimize the size of the payloa d
4.Response optimization: Only provide the fields that are requ ired, not the complete products.

Example Technology Stack (with Code Concepts):

Frontend:

Next.js (React framework) for building performant user interfaces with features like static site generation (SSG) and server-side rendering (SSR).

Backend/API:

- **Framework:** Node.js with Express.Js or a GraphQl server.
- **Authentication:** Passport.js for handling authentication.
- **Validation:** Zod for defining and validating request schema s, ensuring data integrity and security.

JavaScript

```
// Example Zod schema for product filter request
import { z } from 'zod';

const ProductFilterSchema = z.object({
  category: z.string().optional(),
  max-Price: z.number().optional(),
  // ... other filter fields
});

type Product Filter = z.infer<typeof ProductFilterSchema>;
```

- **Database:** PostgreSQL or MongoDB, with appropriate indexes on product attributes.
- **Caching:** Redis for caching query results and product data.
- **CDN:** Cloud flare or AWS Cloud-front for serving static assets.

Verification of Security:

1. Validation of Input :

Clean the filter inputs (brand names, price ranges, and category IDs) to avoid: Injection of SQL Injection of NoSQL Injection of Elasticsearch queries
Type enforcement: for example, the price must be numeric and the category ID must be an integer.

2. Control of Access :

Make sure that filters do not reveal products that are prohibited (e.g., secret SK Us, unreleased products). Verify user-
specific restrictions, such as those that limit access to wholesale products to sp ecified accounts.

3. Abuse Protection & Rate Limiting:

Avoid filter spam attacks, in which bots attempt to flood databases with million s of filter combinations.
Include a CAPTCHA or impose restrictions on odd query patterns.

4. Exposure of Data:

Make that sensitive fields (such as supplier costs and unpublished stock levels) are not exposed by the product API. In API responses, enforce field whitelisting.

5. Authentication & CORS:

Verify the CORS policy, which only permits trusted domains.
Authentication tokens are needed for customized filters.

4. Testing Enhancement:

1.Filtering Capabilities :

1. When a filter is used, the right products appear.
2. Several filters (e.g., category + price + color) cooperate (validation u sing AND/OR logic). The catalog is correctly reset when filters are cleared.

3. Filters remain active during page refreshes, sorting, and pagination.

2. Types of Filters :

Price ranges: Slider functions, edge values are included, and manual i nput is acceptable. Classifications and subcategories:
There are no missing elements in the hierarchy.
Features: (size, color, brand, etc.) - allows for several choices.
Stock/availability: toggle between in-stock and out-of-stock.
Ratings/Reviews: Using a rating filter yields accurate results.

5. UX & Usability:

Filter reaction time (items load in the desired number of seconds).
The filters (position, labeling, sticky sidebar/mobile drop down) are easily accessible. Mobile responsiveness (filter drop downs and accordion). Each active filter is detachable and easily seen.

6. Cases at the Edge:

"Clear Filters" CTA + helpful empty state message => No results found.
Filters that conflict (size S, for example)Filters that conflict (size S + XXL only wh en unavailable, for example).
Check for performance issues and lazy loading in a large product catalog.
unique input or special characters in search-based filters.

6. Incorporation :

Compatible with search (filter + search terms).
After filtering, it works with sorting (e.g., "Price low to high").
Compatible with wishlist and cart (cart state shouldn't be reset by filtering).

7. Analytic & Performance:

Determine the loading speed both with and without filters.
Keep track of the filters that clients really use.
Verify the effects on SEO if filter parameters (canonical, index/no-index) are present in URLs.

6.Deployment(Netlify,vercel or cloud platform):

1.Development(The deployment of Netlify):

Ideal for client-side filters in static or Jam stack catalogs. Steps:
Upload your project to GitHub/Git-lab using React, Vue, Angular, or simple HTML/JS.
Connect the repository to Netlify, and it will automatically build and deploy.
It is optional to include Netlify Functions for lightweight API such as /api/product s.
Product data should be kept in: JSON documents (for tiny catalogs)
CMS without a head (Strapi, Sanity, Contentful) External Database API
An example of a URL HTTP://your-catalog.netify.app

If you want minimum upkeep and simplicity, use this.

2. Deployment of Vercel :

Ideal for Next.js applications where server-side rendering and SEO are important. Steps:
Upload the Next.js/React project to GitHub/Git Lab.
Link the repository to Vercel and use CI/CD for auto deployment.
For the logic behind product filtering, use API Routes in /pages/API.
To ensure that filters update as the catalog changes, enable Incremental Static Regeneration (ISR).

Connect to a database (Firebase, MongoDB Atlas, or Supabase) if you'd like. An example of a URL
https://yourcatalog.vercel.app

3.Platforms for the cloud:

(AWS, GCP, Azure) Ideal for extensive catalogs (over 100,000 items, sophisticated filtering, and business traffic). Steps (for instance, AWS): Front-end → Use S3 + Cloud-front to host the React/Next.js build.
Backend API → Use ECS (containers) or AWS Lambda (serverless) to provide product filter APIs.
Database → For products, use DynamoDB or RDS (PostgreSQL/MySQL).
Search/Filters → Include Algolia, MeiliSearch, OpenSearch, and Elasticsearch.
CI/CD: Make use of GitHub Actions or AWS Code-pipeline. An example of a URL
https://catalog.your-domain.com

Use this if you require search engine integration, scalability, and custom filtering logic. Scale-Based Recommendations for Deployment :

Small Project/Dem → Neltify (easy, quick, and free).

Medium Project (needs SSR and SEO) -> Vercel (optimized for Next.js).

Enterprise/Large Project => Cloud Platform (search + scalable backend).

Here is a guide using Next.js with Vercel and a separate example for a static site on Netlify.

Option 1: Next.js on Vercel:

Vercel is the platform created by the developers of Next.js, making it a powerful and straightforward choice for dynamic or server-rendered product catalogs.

Code for a sample Next.js product catalog:

This example uses a static JSON file for products, which you could replace with a real API or database later.

File: `pages/index.js`

javascript

```
import { useState } from 'react';import Head from 'next/head';

const products = [

 { id: 1, name: 'Laptop', category: 'Electronics', price: 1200 },

 { id: 2, name: 'T-shirt', category: 'Apparel', price: 25 },

 { id: 3, name: 'Smartphone', category: 'Electronics', price: 800 },

 { id: 4, name: 'Jeans', category: 'Apparel', price: 50 },

 { id: 5, name: 'Keyboard', category: 'Accessories', price: 75 },

];

export default function Home() {

 const [filter, setFilter] = useState('');

 const [filteredProducts, setFilteredProducts] = useState(products);



 const handleFilterChange = (e) => {

  const value = e.target.value.toLowerCase();

  setFilter(value);

  const newProducts = products.filter(p =>
p.name.toLowerCase().includes(value) ||
p.category.toLowerCase().includes(value));

  setFilteredProducts(newProducts);

 };



 return (
```

```
  <div style={{ padding: '20px', fontFamily: 'sans-serif' }}>
    <Head>
      <title>Product Catalog</title>
    </Head>
    <h1>Product Catalog</h1>
    <input
      type="text"
          placeholder="Filter by name or category..."
      value={filter}
      onChange={handleFilterChange}
      style={{ marginBottom: '20px', padding: '8px', width: '300px' }}
    />
    <div style={{ display: 'grid', gridTemplateColumns: 'repeat(auto-fill,
minmax(250px, 1fr))', gap: '20px' }}>
      {filteredProducts.map(product => (
        <div key={product.id} style={{ border: '1px solid #ccc', padding: '15px',
borderRadius: '8px' }}>
          <h3>{product.name}</h3>
          <p><strong>Category:</strong> {product.category}</p>
          <p><strong>Price:</strong> ${product.price}</p>
        </div>
      ))}
    </div>
  </div>
 );
```

```
}
```

Use code with caution.

Deployment steps for Vercel:

1. **Set up your repository**: Push your Next.js project to a Git repository (e.g., GitHub, GitLab).
2. **Log in to Vercel**: Sign up or log in to Vercel.
3. **Import your project**:

1. Click "Add New..." and select "Project."
2. Connect your Git provider and import the repository containing your Next.js project.

4. **Configure and deploy:**

Vercel will automatically detect that you are using Next.js and pre-fill the correct settings.

1. Click "Deploy."
2. Vercel will build your application and provide a live URL upon completion.

Option 2: Static site with Netlify:

Netlify is excellent for deploying static front-end projects, including those built with React or plain HTML/CSS and JavaScript.

Code for a sample static product catalog

This example uses a single `index.html` file with JavaScript to handle the filtering.
File: `index.html`

```
html
```

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Product Catalog</title>
    <style>
        body { font-family: sans-serif; padding: 20px; }
        .product-grid { display: grid; grid-template-columns: repeat(auto-fill, minmax(250px, 1fr)); gap: 20px; }
        .product-card { border: 1px solid #ccc; padding: 15px; border-radius: 8px; }
        input[type="text"] { margin-bottom: 20px; padding: 8px; width: 300px; }
    </style>
</head>
<body>
    <h1>Product Catalog</h1>
    <input type="text" id="filterInput" placeholder="Filter by name or category...">
    <div id="product-grid" class="product-grid"></div>

    <script>
        const products = [
            { id: 1, name: 'Laptop', category: 'Electronics', price: 1200 },
            { id: 2, name: 'T-shirt', category: 'Apparel', price: 25 },
            { id: 3, name: 'Smartphone', category: 'Electronics', price: 800 },
            { id: 4, name: 'Jeans', category: 'Apparel', price: 50 },
            { id: 5, name: 'Keyboard', category: 'Accessories', price: 75 },
```

```
    ];

    const productGrid = document.getElementById('product-grid');
    const filterInput = document.getElementById('filterInput');

    const render-products = (productsToRender) => {
        product-grid.inner-HTML = '';
        productsToRender.forEach(product => {
            const card = document.createElement('div');
            card.className = 'product-card';
            card.innerHTML = `
                <h3>${product.name}</h3>
                <p><strong>Category:</strong> ${product.category}</p>
                <p><strong>Price:</strong> $${product.price}</p>
            `;
            product-grid.append-child(card);
        });
    };

    filterInput.addEventListener('input', (e) => {
        const value = e.target.value.toLowerCase();
        const filtered-products = products.filter(p => p.name.to-
lowercase().includes(value) || p.category.to Lowercase().includes(value));
        renderProducts(filteredProducts);
    });
```

```
    document.addEventListener('DOMContentLoaded', () => {
        renderProducts(products);
    });
  </script>
</body>
</html>
```

Deployment steps for Netlify:

Set up your repository: Push your project files (`index.html` and any other assets) to a Git repository.

1. **Log in to Netlify**: Sign up or log in to Netlify.
2. **Import your project**:

1. Click "Add a new site" and select "Import an existing project".
2. Connect your Git provider and choose the repository.
3. **Configure and deploy**:

1. **Build command:**

Leave blank if you only have an `index.HTML` file. If you used a framework like React, use `npm run build`.

2. **Publish directory:**

Use `./` for a static site. Use `build/` for a React app.

Click "Deploy site."

3. **Alternatively, use drag and drop:**

For static sites, you can build your project locally and then simply drag the build folder onto the Netlify dashboard.

Option 3: Cloud provider (e.g., AWS Amplify):

For more complex applications or to leverage a wider range of back-end services, a cloud provider is a robust option. AWS Amplify is a popular choice that simplifies the process for web and mobile developers.

Deployment steps for AWS Amplify

1. **Initialize your project:**

Set up your project locally using the Amplify CLI and create your front-end code with filters. Amplify can also help set up a GraphQL API for your product data.

2. **Connect your repository:**

In the AWS Management Console, navigate to AWS Amplify.

Select "Host web app" and choose your Git provider.

3. **Configure the build**:

Amplify will automatically detect your front-end framework (e.g., React, Next.js) and configure the build settings.

Review and confirm the build command and output directory.

4. **Deploy:**

Click "Save and deploy."

Amplify will build your application and set up continuous deployment, updating your site with every push to your repository.