COLLEGE CODE:8203

COLLEGE NAME:A.V.C.COLLGE OF ENGINEERING

DEPARTMENT: INFORMATION TECHNOLOGY

STUDENT NM-ID: 621EA8E6F9108F8B0C5426C6E0559F56

ROLL NO:820323205062

DATE:03.10.2025

COMPLETED THE PROJECT NAME

AS PHASE _TECHNOLOGY

PROJECT NAME: **PRODUCT CATALOG WITH MANGODB**

SUBMITTED BY

NAME:LAKSHMI DHARANI S

MOBILE NO: 9342641989

Additional Features (Product Enrichment)

Advanced Catalog Search and Filtering Mechanics

The foundation of any good catalog is discoverability. This goes beyond simple field-by-field matching.

MongoDB Text Search Implementation: This feature leverages **MongoDB's native text indexes** on fields like `name` and `description`. Unlike regex-based searches, text search allows for:

- **Stemming:** Searching for "running shoes" finds results for "run," "ran," and "runner."
- **Stop Word Removal:** Ignoring common words like "the," "a," and "is" to focus on relevant keywords.
- **Scoring and Ranking:** Results are returned with a relevance score, allowing the application to display the most pertinent products first, using the `{$meta: "textScore"}` field in the aggregation pipeline.

Faceted Navigation (Multi-Filter): This is implemented on the front-end by managing multiple states for different filter types (e.g., categories, brands, colors). The Node.js/Express backend must construct complex MongoDB queries using the `$and` operator to combine these criteria, allowing users to filter by "Electronics" **AND** "Brand X" **AND** "Price Range Y."

Robust User Authentication and Role Management (RBAC)

Security for admin actions must be impenetrable.

- **JWT and Refresh Tokens:** The Express backend uses **JSON Web Tokens (JWTs)** for session management. The access token, which grants access to protected routes, is short-lived (e.g., 15 minutes).
- The refresh token is long-lived and stored securely (e.g., as an HTTP-only cookie). When the access token expires, the client sends the refresh token to a dedicated endpoint (`POST /auth/refresh`) to obtain a new access token without requiring a full login, enhancing both security and UX.
- **Middleware Implementation:** Authorization is enforced using **Express middleware** on the server. Before any `POST`, `PUT`, or `DELETE /products` request reaches the controller logic, the middleware checks: 1) if a valid access token is present, and 2) if the decoded token's payload contains a `role` field equal to `'admin'`. If either check fails, a `403 Forbidden` response is sent.

Product Reviews and Ratings System

This feature requires careful schema design and API implementation to handle concurrent user inputs.

- **Mongoose Sub-documents:** The `Product` schema is extended to include an array of sub-documents for `reviews`. Each review sub-document contains: `reviewerId (ref: 'User')`, `rating (Number, 1-5)`, `comment (String)`, and `createdAt (Date)`.
- **Server-Side Aggregation:** Upon successful submission of a new review, a post-save hook or a dedicated service function triggers a MongoDB aggregation on the `Product` collection. This pipeline calculates the new average rating and the total number of reviews (`numReviews`), and updates the main product document in a single, atomic operation to ensure data consistency.

UI/UX Improvements (React Front-end)

The focus is on perceived performance and application polish, which heavily relies on the React front-end.

1. Accessibility (A11Y) and Semantic HTML

Accessibility isn't optional; it's a requirement for modern applications.

- **Keyboard Navigation:** Ensuring that all interactive elements (buttons, links, filters, form fields) are navigable using the **Tab key** and that modals can be closed using the **Escape key**. Focus styles must be visually clear.
- **ARIA Roles:** Using **ARIA (Accessible Rich Internet Applications)** attributes to clarify the purpose of non-standard HTML elements. For instance, using `role="status"` for loading messages or `aria-live="polite"` for dynamic content updates (like filter results) so screen readers are alerted.
- **Form Field Labels:** Associating all form inputs (e.g., the search bar) with a descriptive `<label>` using the `for` and `id` attributes.

2. Enhanced Data Display and Load Handling

A professional application never leaves the user waiting for a blank screen.

- **Skeleton Loading Screens:** Instead of displaying a generic "Loading..." spinner, the UI shows a simplified, unpopulated version of the product cards, known as **skeleton**

components. This gives the illusion of faster loading by immediately showing the screen structure, significantly improving the perceived performance.

- **State Management for Filters:** Implementing an **optimistic UI update** for filters. When a user clicks a filter, the front-end immediately shows the filter as "active" while simultaneously making the API call. If the call succeeds, the data updates. If it fails, the filter state reverts, providing a highly responsive feel.

API and Backend Enhancements (Node.js/Express)

Backend refinement is key to scalability and resilience.

1. Robust Query Parameter Standardization

The Node.js server acts as an intelligent translator between the client's HTTP request and MongoDB's query language.

- **Deep Filtering Logic:** Implementing a custom query parser to handle complex, nested filtering criteria sent via URL. For example, converting the URL parameter `price[gte]=100&price[lte]=500` into the MongoDB query object: `{"price": { "$gte": 100, "$lte": 500 }}`. This is typically achieved using a utility library or custom middleware that analyzes and transforms the request object's `req.query`.
- **Pagination Headers:** Instead of relying on the client to guess the total number of products, the `GET /products` API response includes custom HTTP headers (e.g., `X-Total-Count`, `X-Total-Pages`) to inform the client about the overall data set size, enabling accurate pagination controls on the front-end.

2. Enhanced Logging and Monitoring

A production app needs to be observable.

- **Centralized Logging:** Using a professional logging library (e.g., **Winston**) configured to write structured logs (JSON format). These logs track key events: request start/end times, user ID for admin actions, API endpoint called, and any errors. In deployment, these logs are pushed to a centralized service like **CloudWatch** or **Loggly**.
- **Health Check Endpoint:** Implementing a simple `/health` endpoint that returns a `200 OK` status only if the Express server is running **and** the MongoDB connection is active. This is vital for load balancers and container orchestration systems to manage application uptime.

Performance & Security Checks

These are the non-functional requirements that validate the system's fitness for production.

1. Performance Optimization: Caching and Indexes

Speed hinges on reducing database queries and response size.

- **MongoDB Indexing Deep Dive:** Verifying that **compound indexes** are created for common filter and sort combinations.
- For example, if users frequently search by `category` and sort by `price`, an index on `{"category": 1, "price": -1}` is created. The MongoDB `explain()` function is used to analyze and optimize the performance of the most critical queries.
- **Response Compression (Gzip/Brotli):** Using the `compression` middleware in Express to automatically compress JSON response payloads.
- This drastically reduces the size of data transmitted over the network, especially for large lists of products, accelerating client loading times.

2. Security Audits: OWASP Top 10 Mitigation

Security is addressed proactively.

- **NoSQL Injection Prevention:** Since the application uses MongoDB, a critical step is validating all input and ensuring no user input is concatenated directly into a query.
- This is specifically mitigated by using Mongoose's built-in query sanitization and dedicated libraries like `express-mongo-sanitize` to strip prohibited characters (like `$` or `.`) from user input keys.
- **CORS Configuration:** Rigorously configuring **Cross-Origin Resource Sharing (CORS)** middleware to only allow requests from the expected front-end domain (e.g., `https://myproductcatalog.vercel.app`), preventing unauthorized third-party websites from making malicious requests.

Testing of Enhancements

Testing validates that the application works as intended under stress.

1. Integration and End-to-End (E2E) Testing

Moving from isolated unit tests to full system checks.

- **Test Environment:** Setting up a dedicated testing database instance (e.g., a **sandbox MongoDB Atlas cluster**) that is separate from both development and production databases.
- **E2E Scenarios (Cypress/Playwright):** Implementing E2E tests to simulate full user flows:
  - "Guest User browses, applies advanced filter, and successfully sorts the results."
  - "Admin User logs in, navigates to the dashboard, and successfully posts a new product." This confirms that both the React UI and the protected Express API work together seamlessly.

2. Load and Stress Testing

Determining the application's actual capacity.

- **Throughput Benchmarking:** Using tools like **Artillery** to measure the **Requests Per Second (RPS)** the `/products` endpoint can handle before latency significantly increases. The goal is to set a performance baseline and identify the server/database scaling limits.

Deployment Deep Dive (Netlify/Vercel and Cloud Platform)

This is the final execution step, making the code live.

1. CI/CD Pipeline with GitHub Actions

Automation ensures zero-downtime deployment.

- **Workflow Definition:** A GitHub Actions YAML file is defined with a workflow triggered by a push to the `main` branch.
    1. **Build Phase:** Install dependencies, run unit/integration tests, and perform the React build (`npm run build`).
    2. **Frontend Deployment:** Push the generated static files (the React build) to **Netlify/Vercel**.
    3. **Backend Deployment:** Use a deployment script to push the Node.js code to the chosen PaaS (e.g., **Render** or **Heroku**).
    4. **Database Connection:** Ensure the production backend is configured to connect to the dedicated **MongoDB Atlas** production cluster, using secure production environment variables.

2. Infrastructure as Code

For cloud platforms, this ensures repeatable, consistent deployment.

- **Configuration Files:** Using configuration files (e.g., `vercel.json`, `app.yaml`) to specify production settings, caching rules, environment variable injection, and custom domain configuration, guaranteeing that the environment is consistent across deployments.