# Features list

## Global search bar

* This component enables the User to search the entire collection of items on the website by name.
* It consists of a *DropdownFilter* and an *Input* component – no autocomplete.
* The *DropdownFilter* contains only the **TOP-LEVEL Categories** – no subcategories.
* The default selection is **All**.
* Upon keypress *Enter*, route the User to a *./products/<category\_filter>/<searchbar\_input>* dynamic route with products fetched by category and search value. (atlas search recommended). List products relevant to search parameters.

## CMS (content/product-management-system)

* This feature is an admin-only route inside the application, also visible by the admin only, where admin is a **role**.
* It consists of a relatively simple Form with the following fields:
  + productName: string
  + category: string
  + price: double/decimal
  + discount?: double/decimal
  + images: string[] – *preferably url for images on edge-store (AWS S3 bucket later?)*
  + stock: integer
  + description: string
  + attributes: object (object with more product-specific properties -> mongo schemas) – form with *add field* button for adding attributes
* Upon *submit* add the product to the database.

## Sidebar

* Fully dynamic with nested dropdowns
* Must add SidebarGroup to encapsulate top-level categories

User accounts

Profile page

Shopping cart

Wishlist

Payment system

Most popular items / recommendations on landing page with flashy prices

Products by brand search

# Entities

## Product (Item)

* Properties:
  + \_id : ObjectID (string)
  + productId: string ***UNIQUE***
  + productName: string
  + category: string
  + price: double/decimal
  + discount?: double/decimal
  + images: string[]
  + stock: integer
  + description: string
  + attributes: object (object with more product-specific properties -> mongo schemas)

## Category

* Properties:
  + name: String
  + subcategories: Array<Category>
  + image?: string

## User

* Properties:
  + Id : Integer
  + Email : String
  + Username: String
  + Password? : String
  + PhoneNumber? : String
  + Role: “customer” | “admin”

# Development

## Source Control

* Practice **monorepo** style on git
* Project structure:

/me-commerce

/frontend (Next.js app)

/backend (SpringBoot app)

/README.md

/docs

## Initial development roadmap

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | * Phase | * Focus | * Why | | * Phase 1 | * User Authentication (Sign up/in/out, secure sessions) | * Core foundation | | * Phase 2 | * Product Management (Add/Edit/Remove products) | * You can't sell without items | | * Phase 3 | * Today's Deals / Featured Products section | * Landing page appeal | | * Phase 4 | * Search & Filters (Category, Price Range) | * Essential for user experience | | * Phase 5 | * Cart & Checkout flow (even if only "Add to Cart" initially) | * Start "feeling" like a real ecommerce | | * Phase 6 | * Orders management (backend only at first) | * Needed to process orders eventually | |

# Deployment

* Deploy the Frontend to one domain: <https://me-commerce.com>
* Deploy the Backend to a different domain: <https://api.me-commerce.com>
* Rough roadmap for the future (GPT response/advice):

**🧩 Step-by-Step Plan:**

**1. Domain Name**

* Buy or use an existing domain, e.g., me-commerce.com.
* Manage it through **Route53** (AWS DNS service).

**2. MongoDB Database**

* Use **MongoDB Atlas** (Free or Paid tier).
  + Reliable, auto-scaling, backed up, and **accessible from AWS easily**.
  + Keep DB access *restricted by IP / security group* for security.

(MongoDB could technically be hosted on an EC2 server, but Atlas is way less headache.)

**3. Backend Deployment (Spring Boot)**

* Launch an **EC2 instance** (Amazon Linux or Ubuntu).
  + Install Java (OpenJDK 17+).
  + Install Nginx (as a reverse proxy, optional).
  + Deploy your Spring Boot app (build .jar or .war).
* Set up **SSL Certificate** using **ACM** (AWS Certificate Manager).
  + Use **Elastic IP** so your IP doesn’t change.
* Open necessary ports (80 for HTTP, 443 for HTTPS, custom app port if needed internally).
* Your API will be reachable at:  
  https://api.me-commerce.com

**Extra Tip**:  
You could also **Dockerize** your backend and run it via **Amazon ECS** later if you want more scalability!

**4. Frontend Deployment (Next.js 15)**

**Options**:

* **Cheapest/Easiest**: Deploy static export to **S3 + CloudFront**.
  + You can run next build && next export to generate static files.
* **More dynamic (Recommended)**: Deploy the Next.js app on **Vercel** or **Amplify Hosting** (AWS).
  + Vercel is built by the creators of Next.js — very good for SSR.
  + Amplify Hosting is AWS-native, supports SSR and API integrations.

**If you go with EC2** for the frontend too:

* Spin up a second EC2 instance (Node.js installed).
* Deploy Next.js like a normal Node server (next start).

**5. Networking**

* **Route53**:
  + me-commerce.com → points to Frontend.
  + api.me-commerce.com → points to Backend.
* **Security Groups**:
  + Frontend EC2: Open ports 80 (HTTP) and 443 (HTTPS).
  + Backend EC2: Open ports 443 and private app ports if needed (only allow traffic from frontend EC2 or internet if API public).
* **Load Balancer (Optional, for production)**:
  + You can put your Spring Boot app behind an **Application Load Balancer (ALB)** for high availability.

**6. SSL (HTTPS everywhere)**

* Use **AWS ACM** to issue free SSL certificates.
* Attach SSL to your Load Balancer or directly configure it with Nginx.