COLLEGE CODE : 8206

COLLEGE NAME : ARASU ENGINEERING COLLEGE

DEPARTMENT : ARTIFICIAL INTELLIGENCE & DATA SCIENCE

STUDENT NM ID :C5C3631B76BE12FA3DF404E3A81E8742

Completed the project named as :

IBM -NJ- PRODUCT CATALOG WITH MONGO DB

SUBMITTED BY,

M.FARHATH ZAFIN

820623243016

TEAM MEMBERS

R.GOWRI,

820623243017

S.HARITHA

820623243019

PHASE_1 Product Catalogue with MongoDB

1. Problem Statement

E-commerce businesses often struggle with managing large product catalogs across multiple categories, variants, and price ranges. A centralized product catalogue service is needed that:

- Stores flexible product data (name, description, images, price, stock).
- Supports scalability (thousands to millions of products).
- Enables quick search/filtering (by category, price, tags).
- Integrates easily with inventory, orders, and recommendation engines.

MongoDB’s flexible schema makes it ideal for managing dynamic product attributes and variants.

2.user

- End Customers – Browse/search products, filter by categories, view details.
- Admin/Managers – Add, update, or remove products, manage stock & categories.
- Business Owners – Monitor product availability, pricing, and catalog growth.
- Developers – Extend APIs, build integrations with frontend/other services.

3. User Stories

- As a customer, I want to search and filter products so I can quickly find what I want.
-As a customer, I want to view product details (price, stock, variants, images) before purchasing.
-As an admin, I want to add and edit products so that the catalog is always up-to-date.
- As an admin, I want to track stock levels so I know when to reorder.
- As a business owner, I want insights into which categories/products sell most.

4. MVP Features

Core Product Management

- Create, read, update, delete (CRUD) products.
- Categories, tags, and brand support.
- Variants (color, size, etc.).

Search & Browse

- Filter by category, brand, price.
- Text search by product name/tags.

Inventory Tracking

- Stock levels per SKU.
- Auto decrement on purchase (API simulation).

Admin Features

- Role-based API access (admin vs. customer).

5. Wireframes / API Endpoints

Wireframes (Text-based Sketch):
Customer View → Homepage → Categories →

Product List → Product Details
Admin View → Dashboard → Add/Edit Product → Manage Inventory

REST API Endpoints:

Products:
- POST /api/products → Add product (admin)
- GET /api/products → List products with filters
- GET /api/products/:id → Get single product details
- PUT /api/products/:id → Update product
- DELETE /api/products/:id → Delete product

Inventory:
- PATCH /api/products/:id/stock → Update stock
- GET /api/inventory/low-stock → Get products low in stock

Categories:
- POST /api/categories → Add category
- GET /api/categories → List categories

6. Acceptance Criteria

- Product CRUD works: Admin can add, edit, delete products and see changes reflected immediately.

- Customer browsing works: Products can be searched by keyword and filtered by category/price.
- Inventory updates correctly: Stock count decreases after purchase simulation.
- Performance: Product queries return within 300ms for up to 10,000 products.
- Validation: Products cannot be created without required fields (name, price, category, stock).

7. KUDU (Key Unique Differentiators)

- MongoDB Flexible Schema → Easily add product attributes without migrations.
- Variant Management → Handle multiple SKUs under one product (color/size).
- Scalable Search → Indexes on category, price, and tags for fast querying.
- API-first Design → Can plug into mobile apps, web apps, or partner integrations.
- Role-based Access → Separate admin/customer endpoints.

PHASE_2-PRODUCT CATALOG WITH MONGODB

Tech Stack Selection

- Frontend: React.js with TailwindCSS
- - React.js helps in building modular, component-based UI.
- - TailwindCSS ensures responsive and modern design with utility-first CSS.

- Backend: Node.js with Express.js
- - Node.js is chosen for its non-blocking event-driven architecture, which is ideal for handling large API requests.
- - Express.js simplifies API creation with routing, middleware, and error handling.

- Database: MongoDB
- - A NoSQL, document-oriented database suitable for flexible schema and hierarchical product data.

- - Mongoose ODM will be used for schema validation, relations, and queries.

- Authentication & Security: JWT
- - JSON Web Tokens (JWT) will be used for secure login and role-based access control (admin vs user).
- - Passwords stored using bcrypt hashing.

- Deployment: Docker + Cloud (AWS/Heroku)
- - Docker for containerization to ensure consistency across environments.
- - Cloud deployment (AWS/Heroku) for scalability.

- Version Control: GitHub/GitLab for CI/CD pipeline integration.

UI Structure / API Schema Design

- UI Structure:
- - Home Page: Displays list of products with search, filters, and categories.

- - Product Details Page: Shows details like images, description, price, and stock availability.
- - Admin Dashboard: Provides product management (Add, Update, Delete).
- - Cart & Checkout Pages (planned for Phase 3).

- API Schema (RESTful endpoints):
- GET /api/products → Retrieve all products with optional query params for filtering & pagination.
- GET /api/products/:id → Retrieve a specific product by ID.
- POST /api/products → Create a new product (Admin only).
- PUT /api/products/:id → Update existing product (Admin only).
- DELETE /api/products/:id → Delete product (Admin only).
- POST /api/auth/login → Authenticate user/admin and return JWT.

Data Handling Approach

- MongoDB Collections:

- - Products: { _id, name, description, price, category, stock, images[], createdAt, updatedAt }
- - Users: { _id, name, email, passwordHash, role (user/admin) }
- - Orders (future scope): { _id, userId, productIds[], totalAmount, status }

- Approach:
- - Use Mongoose schemas for strong typing & validation.
- - Index product name and category fields for faster searches.
- - Implement server-side pagination for product listing APIs.
- - Store images in cloud (AWS S3) or as URLs, not inside MongoDB.
- - Secure API endpoints with JWT middleware.
- - Handle errors centrally with Express error middleware.

Component / Module Diagram

- Frontend Components:
- - ProductList: Displays products in a grid.
- - ProductCard: Shows brief product info.

- - ProductDetails: Full detail page of a product.
- - AdminPanel: CRUD interface for products.
- - Navbar, Footer, Filters, SearchBar, etc.

Backend Modules:

- - Controllers: Handle API requests (ProductController, AuthController, UserController).
- - Services: Contain business logic (ProductService, AuthService).
- - Models: Define MongoDB collections with Mongoose (Product, User, Order).
- - Middleware: JWTAuthMiddleware, ErrorHandler, Logger.

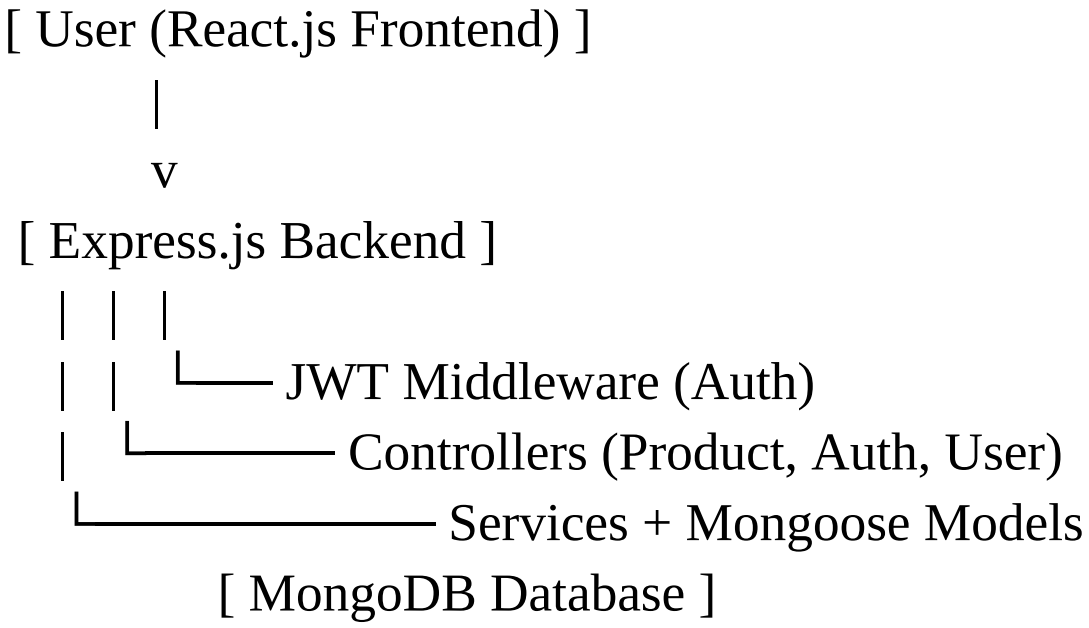
Basic Flow Diagram

- Step 1: User opens React frontend → React fetches data from backend APIs.
- Step 2: API request sent to Express.js server.
- Step 3: Express validates request, authenticates user (if required).
- Step 4: Server interacts with MongoDB using Mongoose.

- Step 5: MongoDB returns requested data to server.
- Step 6: Express sends JSON response back to React.
- Step 7: React renders UI with received data.
- Step 8: For Admin operations, JWT token is validated before performing CRUD actions.

System Flow Diagram:

Below is the block diagram representing the workflow of the Product Catalog system:



Sample Node.js Backend Code

```
// server.js
const express = require('express');
const mongoose = require('mongoose');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');
const app = express();

app.use(express.json());

// MongoDB connection
mongoose.connect('mongodb://localhost:27017/produ
ctCatalog', {
    useNewUrlParser: true,
    useUnifiedTopology: true
});

// User Schema
const userSchema = new mongoose.Schema({
    name: String,
    email: { type: String, unique: true },
    passwordHash: String,
    role: { type: String, enum: ['user', 'admin'], default:
'user' }
```

```
});
const User = mongoose.model('User', userSchema);

// Product Schema
const productSchema = new mongoose.Schema({
    name: String,
    description: String,
    price: Number,
    category: String,
    stock: Number,
    images: [String],
}, { timestamps: true });
const Product = mongoose.model('Product',
productSchema);

// Middleware: JWT Auth
function authMiddleware(req, res, next) {
    const token = req.headers['authorization'];
    if (!token) return res.status(401).json({ error: 'No
token provided' });
    jwt.verify(token, 'secretkey', (err, decoded) => {
        if (err) return res.status(403).json({ error:
'Invalid token' })
```

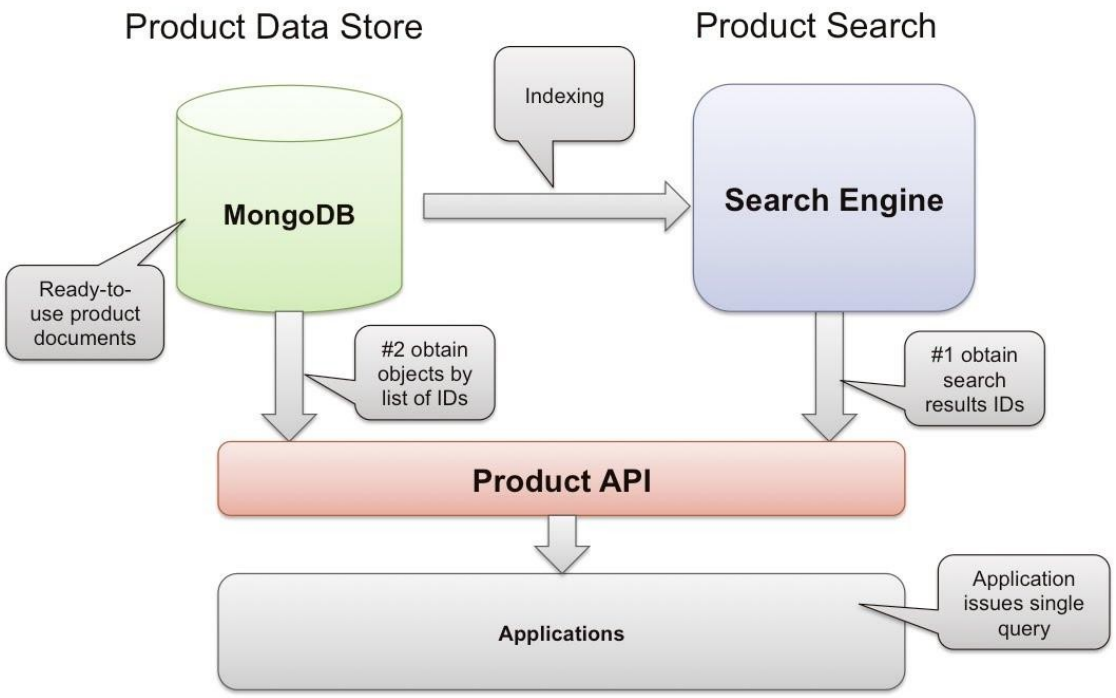
```
 req.user = decoded;
        next();
    });
}

// Register User
app.post('/api/auth/register', async (req, res) => {
    const { name, email, password } = req.body;
    const hash = await bcrypt.hash(password, 10);
    const newUser = new User({ name, email,
passwordHash: hash });
    await newUser.save();
    res.json({ message: 'User registered' });
});

// Login
app.post('/api/auth/login', async (req, res) => {
    const { email, password } = req.body;
    const user = await User.findOne({ email });
    if (!user) return res.status(400).json({ error:
'Invalid email' });
    const valid = await bcrypt.compare(password,
user.passwordHash);
    if (!valid) return res.status(400).json({ error:
```

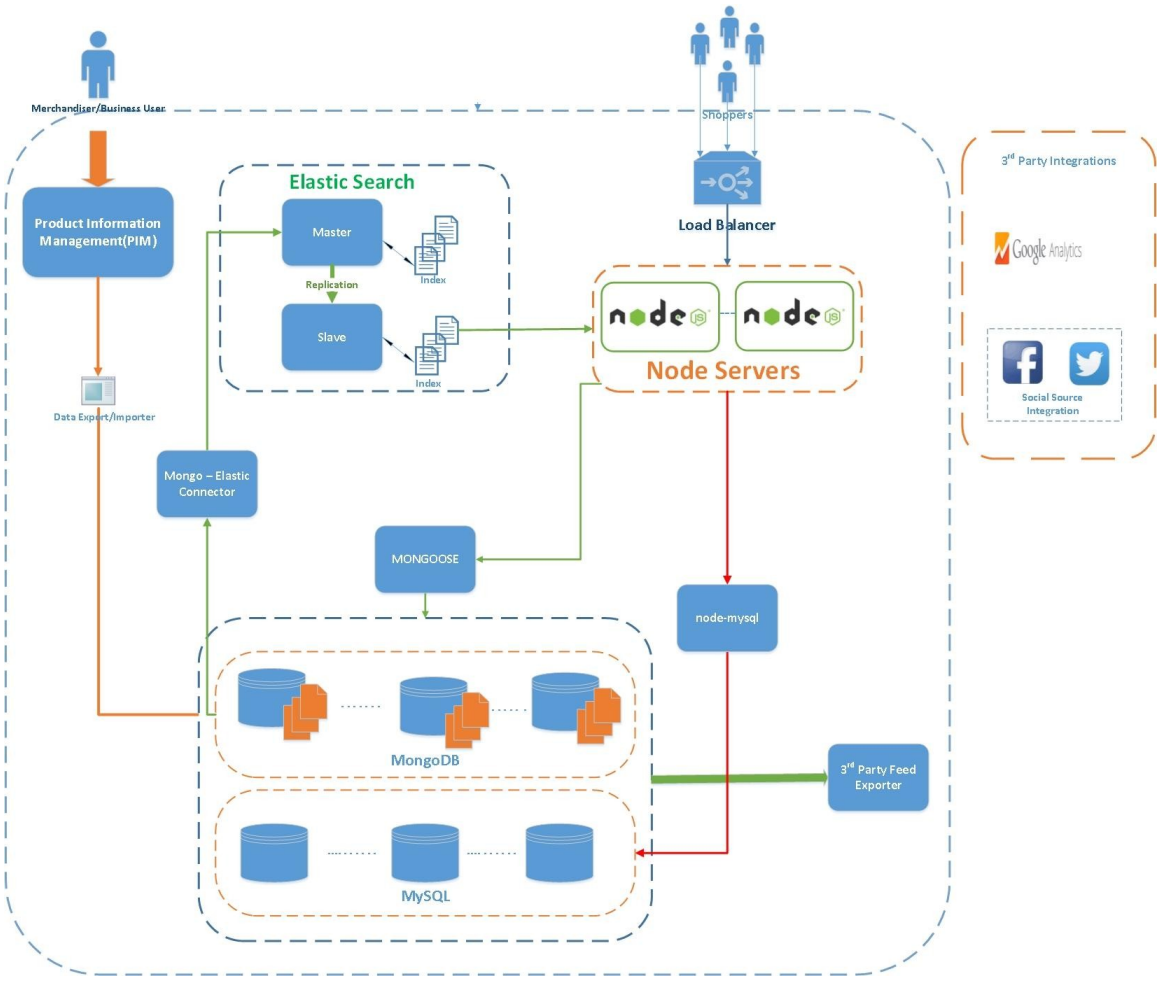
```
'Invalid password' });
    const token = jwt.sign({ id: user._id, role: user.role
}, 'secretkey', { expiresIn: '1h' });
    res.json({ token });
});

// Get All Products
app.get('/api/products', async (req, res) => {
    const products = await Product.find();
    res.json(products);
});

// Get Product by ID
app.get('/api/products/:id', async (req, res) => {
    const product = await
Product.findById(req.params.id);
    res.json(product);
});

// Add Product (Admin only)
app.post('/api/products', authMiddleware, async (req,
res) => {
    if (req.user.role !== 'admin') return
res.status(403).json({ error: 'Forbidden' });
```

```
    const newProduct = new Product(req.body);
    await newProduct.save();
    res.json(newProduct);
});

// Update Product
app.put('/api/products/:id', authMiddleware, async
(req, res) => {
    if (req.user.role !== 'admin') return
res.status(403).json({ error: 'Forbidden' });
    const updated = await
Product.findByIdAndUpdate(req.params.id,
req.body, { new: true });
    res.json(updated);
});

// Delete Product
app.delete('/api/products/:id', authMiddleware, async
(req, res) => {
    if (req.user.role !== 'admin') return
res.status(403).json({ error: 'Forbidden' });
    await Product.findByIdAndDelete(req.params.id);
    res.json({ message: 'Product deleted' });
});
```

app.listen(5000, () => console.log('Server running on port 5000'));

BLOCK DIAGRAM:




Merchandiser/Business User
Shoppers
Product Information Management(PIM)
Elastic Search
Master
Index
Replication
Slave
Index
Load Balancer
Node Servers
3rd Party Integrations
Google Analytics
Social Source Integration
Data Export/Importer
Mongo – Elastic Connector
MONGOOSE
node-mysql
MongoDB
MySQL
3rd Party Feed Exporter

PHASE_3- MVP IMPLEMENTATION FOR PRODUCT CATALOG MONGODB

Introduction

Phase 3 focuses on developing the Minimum Viable Product (MVP) for a Product Catalog System using MongoDB. The goal of this phase is to implement the essential functionalities that allow the catalog to operate efficiently before integrating advanced features. The MVP includes CRUD operations, data persistence, frontend-backend integration, testing, and version control. This ensures that the system is functional, reliable, and ready for future enhancements.

1. Project Setup

- Install Node.js, Express.js, and MongoDB.
- Folder Structure:
  - models/ → MongoDB schemas (Product model)
  - routes/ → API endpoints (productRoutes.js)
  - controllers/ → Business logic and CRUD operations
  - config/ → Database connection and environment variables

- Dependencies: mongoose, dotenv, express, body-parser, cors.
- Use `.env` to store sensitive information like MongoDB URI.
- Setup Express server to listen on a specific port.

2. Core Features Implementation

- Implement CRUD operations: Create, Read, Update, Delete products.
- Search and filter by name, category, price.
- Sort products by price, rating, or other attributes.
- Integrate frontend with backend APIs for real-time interaction.

Sample Program

Server Setup (server.js):

```javascript
const express = require('express');
const mongoose = require('mongoose');
const dotenv = require('dotenv');
const productRoutes =
require('./routes/productRoutes');
dotenv.config();
```

```
const app = express();
app.use(express.json());
mongoose.connect(process.env.MONGO_URI,
{ useNewUrlParser: true, useUnifiedTopology:
true })
  .then(() => console.log('MongoDB connected'))
  .catch(err => console.log(err));
app.use('/api/products', productRoutes);
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Server running
on port ${PORT}`));
```

Product Model (models/Product.js)

```javascript
const mongoose = require('mongoose');
const productSchema = new mongoose.Schema({
  name: { type: String, required: true },
  category: { type: String, required: true },
  price: { type: Number, required: true },
  description: { type: String },
  stock: { type: Number, default: 0 },
  createdAt: { type: Date, default: Date.now }
});
```

```
module.exports = mongoose.model('Product',
productSchema);
```

Routes (routes/productRoutes.js)

```javascript
const express = require('express');
const router = express.Router();
const { getProducts, createProduct,
updateProduct, deleteProduct } =
require('../controllers/productController');
router.get('/', getProducts);
router.post('/', createProduct);
router.put('/:id', updateProduct);
router.delete('/:id', deleteProduct);
module.exports = router;
```

Controllers (controllers/productController.js)

```javascript
const Product = require('../models/Product');
exports.getProducts = async (req, res) => {
   try { const products = await Product.find();
```

```
res.json(products); }
   catch (err) { res.status(500).json({ message:
err.message }); }
};
exports.createProduct = async (req, res) => {
   const product = new Product(req.body);
   try { const newProduct = await product.save();
res.status(201).json(newProduct); }
   catch (err) { res.status(400).json({ message:
err.message }); }
};
exports.updateProduct = async (req, res) => {
   try { const updatedProduct = await
Product.findByIdAndUpdate(req.params.id,
req.body, { new: true });
        res.json(updatedProduct); }
   catch (err) { res.status(400).json({ message:
err.message }); }
};
exports.deleteProduct = async (req, res) => {
   try { await
Product.findByIdAndDelete(req.params.id);
res.json({ message: 'Product deleted' }); }
   catch (err) { res.status(500).json({ message:
```

```
err.message }); }
};
```

3. Data Storage (Local State / Database)

- MongoDB stores persistent product data.
- Schema validation ensures data integrity.
- Frontend local state (React/Redux) handles temporary data such as cart items or filters.
- Ensures synchronization between frontend UI and backend database.

4. Testing Core Features

- Unit Testing: Test individual functions (e.g., product creation).
- Integration Testing: Ensure APIs interact correctly with MongoDB.
- Frontend Testing: Validate UI reflects backend data.
- Tools: Jest, Mocha/Chai, Postman.
- Automated testing ensures functionality after updates.

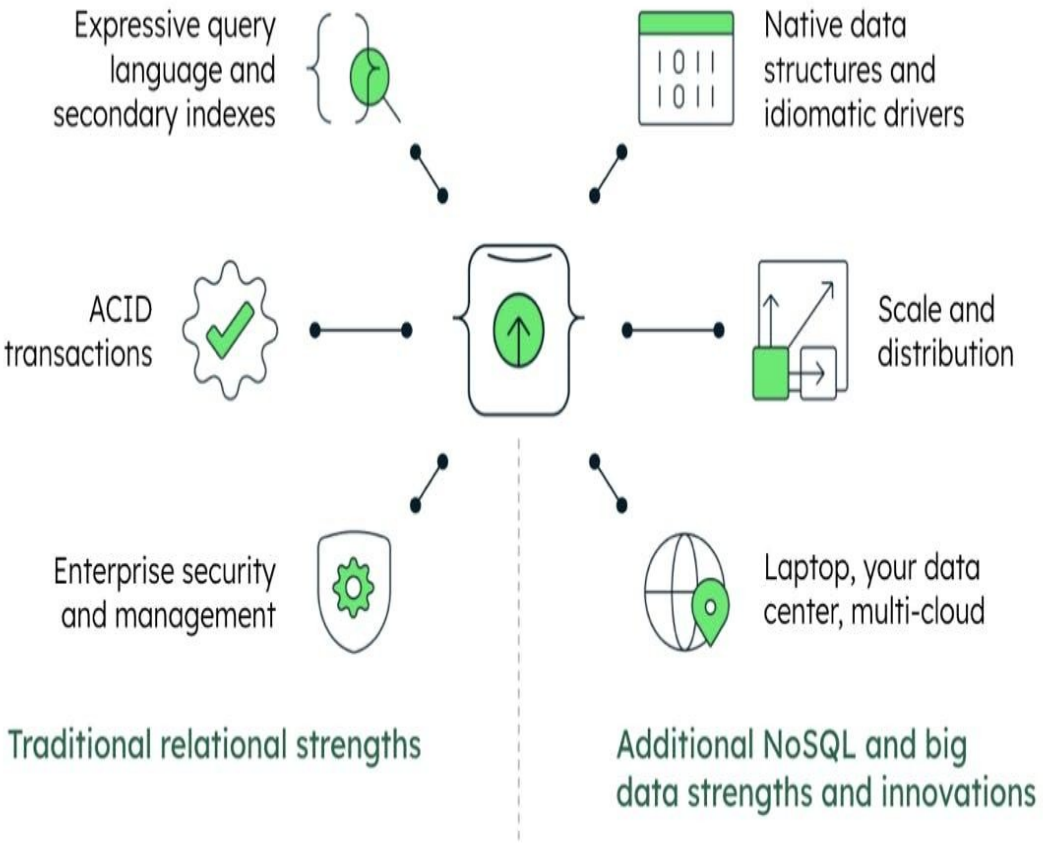
5. Version Control (GitHub)

- Initialize Git repository and commit changes.
- Use branches for features and bug fixes.
- Push changes to GitHub for collaboration and backup.
- Pull requests and code reviews maintain code quality.
- Version history allows tracing and reverting changes.

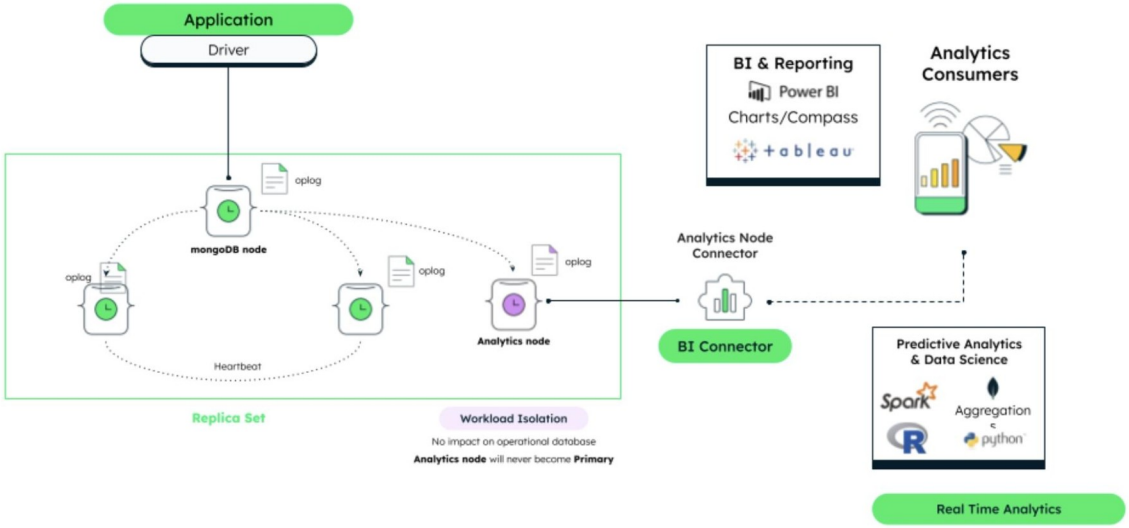
Conclusion

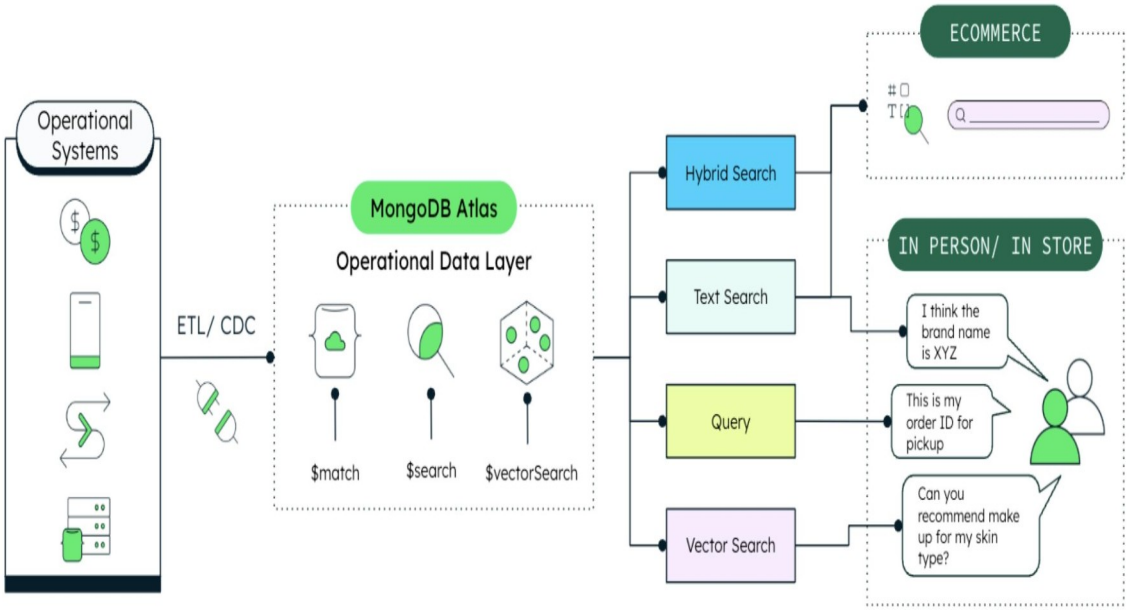
Phase 3 delivers a fully functional MVP of the Product Catalog System. By implementing proper project setup, CRUD functionality, reliable data storage, rigorous testing, and version control, the system is scalable, maintainable, and ready for future enhancements. This structured approach ensures a solid foundation for adding advanced features and improving the product.

BLOCK DIAGRAM:



Replica Set : Workload Isolation





Phase 4 — Enhancements & Deployment (Product Catalog with MongoDB)

Overview

Phase 4 focuses on improving the initial MVP by adding additional features, enhancing the UI/UX, optimizing APIs, and ensuring performance and security. This phase ensures that the product catalog is production-ready and user-friendly.

1. Additional Features

- Search & Filters: Users can search products by name, category, price, or rating.
- Pagination & Sorting: Efficient browsing for large catalogs.
- Wishlist & Cart Integration: Users can save or purchase products easily.
- Notifications: Send alerts for promotions, stock updates, or new products.

Example: Product Search API

```
const express = require('express');
const router = express.Router();
```

```
const Product = require('./models/Product');

router.get('/search', async (req, res) => {
  const query = req.query.q;
  try {
    const results = await Product.find({
      $or: [
        { name: { $regex: query, $options: 'i' } },
        { category: { $regex: query, $options: 'i' } }
      ]
    });
    res.json(results);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

module.exports = router;
```

2. UI/UX Improvements

- Responsive Design: Works on mobile, tablet, and desktop.
- Clean Product Layout: Card or grid view with

images, prices, and ratings.
- Interactive Features: Hover effects, quick view, image zoom, and filters.

Example: React Product Card Component

```
function ProductCard({ product }) {
  return (
    <div className="card p-4 rounded-lg shadow-md">
      <img src={product.image} alt={product.name} className="w-full h-48 object-cover rounded" />
      <h3 className="mt-2 font-bold">{product.name}</h3>
      <p className="text-gray-600">${product.price}</p>
    </div>
  );
}
export default ProductCard;
```

3. API Enhancements

- Optimized Endpoints: Reduce payload and improve speed.
- Advanced Filtering & Sorting: By category, price range, popularity.
- Error Handling & Validation: Ensure correct data input.

Example: Product Schema with Validation

```
const productSchema = new mongoose.Schema({
  name: { type: String, required: true },
  price: { type: Number, required: true, min: 0 },
  category: { type: String, required: true },
  stock: { type: Number, default: 0 }
});
```

4. Performance & Security Checks

- Performance:
  - Index frequently searched fields (name, category).
  - Cache popular products using Redis.
- Security:
  - JWT-based authentication for APIs.

- Input validation to prevent MongoDB injections.
- HTTPS for secure data transfer.

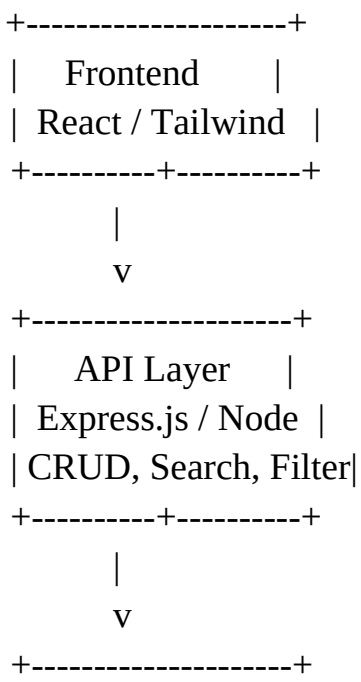
Example: MongoDB Index

```
productSchema.index({ name: 1, category: 1 });
```

5. Deployment

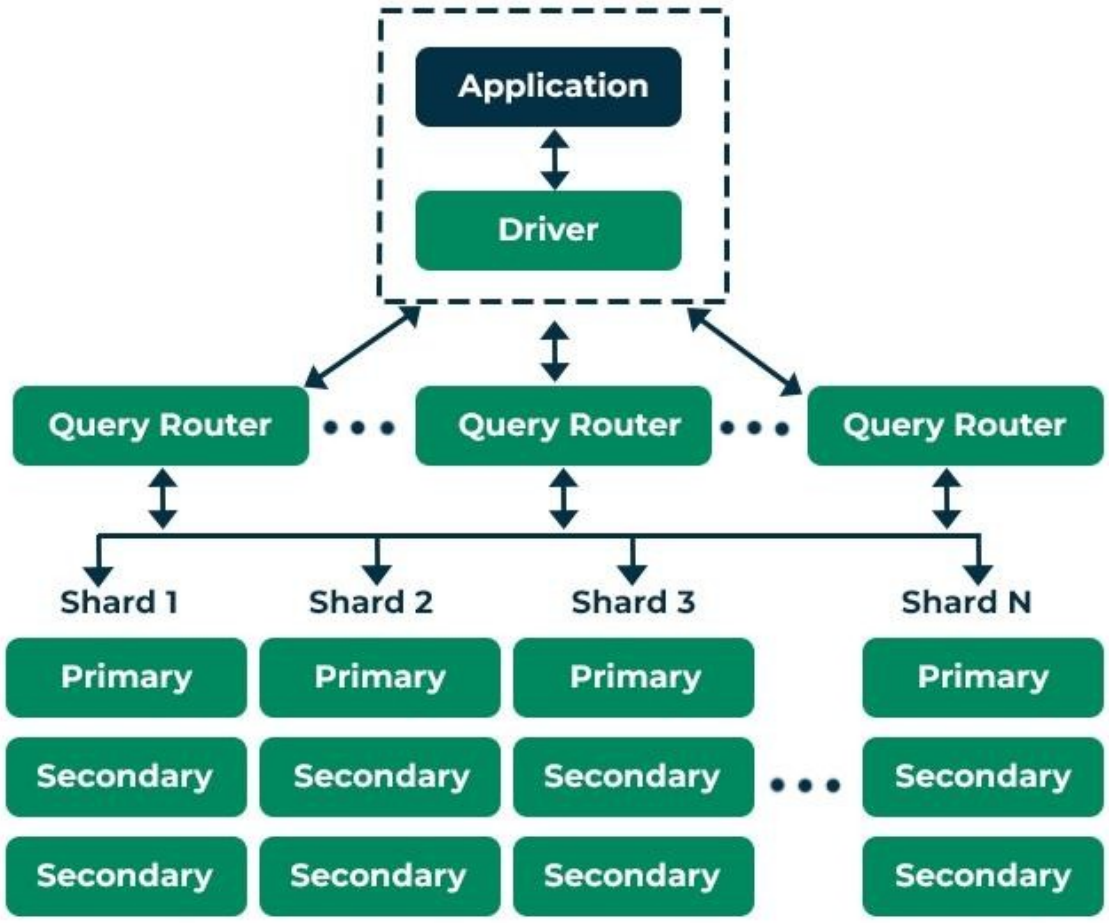
- Use cloud platforms like AWS, Heroku, or DigitalOcean.
- Store sensitive data in environment variables (DB URI, API keys).
- CI/CD with GitHub Actions for automated deployment and updates.

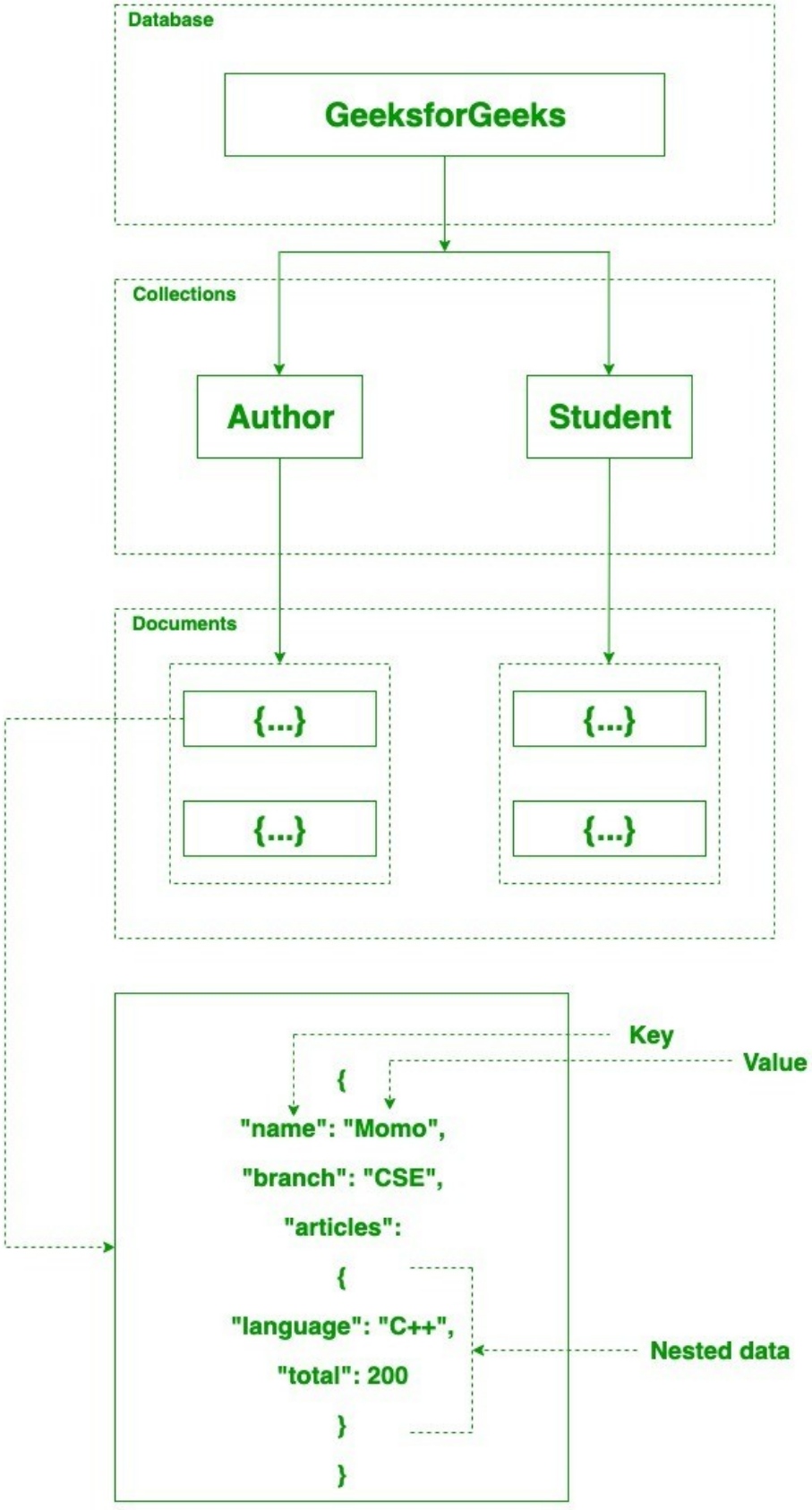
6. System Diagram



```
|      MongoDB        |
| Product Collection  |
| User Collection     |
+---------------------+
```

BLOCK DIAGRAM




Database
GeeksforGeeks
Collections
Author
Student
Documents
{...}
{...}
{...}
{...}
Key
Value
{
"name": "Momo",
"branch": "CSE",
"articles":
{
"language": "C++",
"total": 200
}
}
Nested data

Phase 5 — Project Demonstration & Documentation

Final Demo Walkthrough

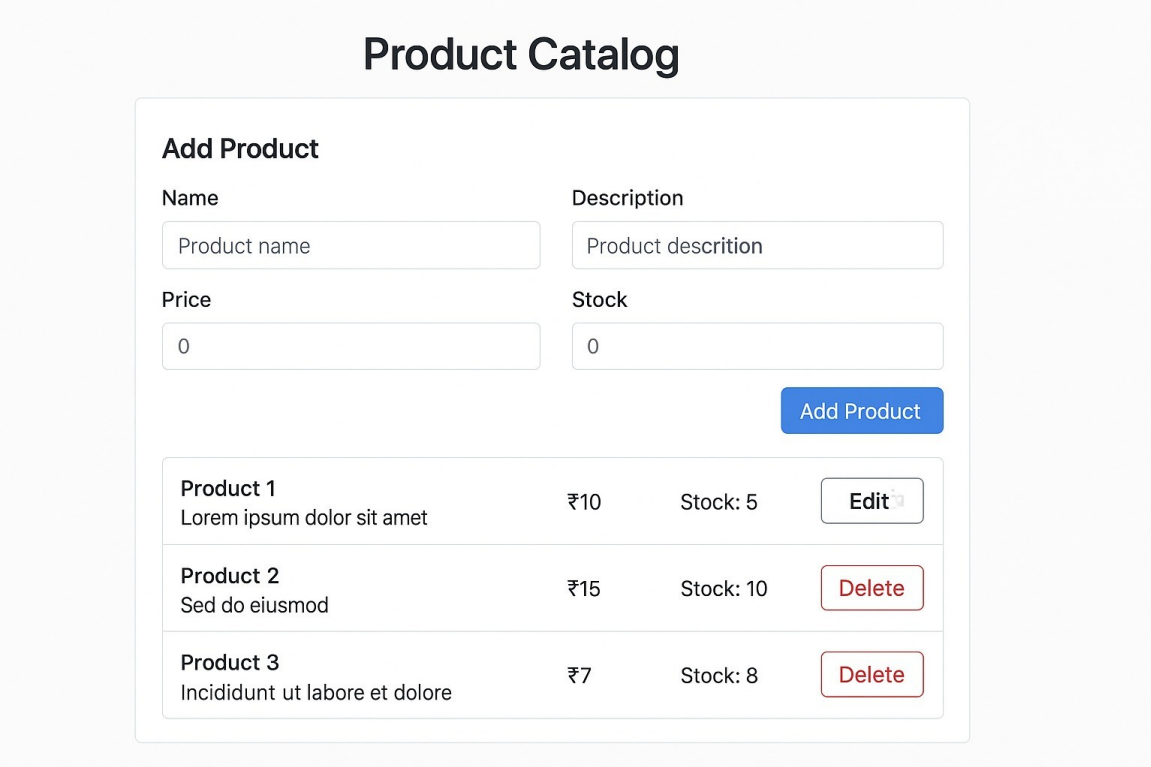
- The final demo walkthrough is the practical demonstration of the entire project.
- It includes showcasing all the features and functionalities of the developed system.
- During the walkthrough, each module is explained step-by-step to highlight how the project meets its objectives.
- The demonstration also focuses on how user interaction, input/output processes, and real-time operations work together.
- It is usually presented to the project guide or evaluation panel for assessment.

Project Report

- The project report is a detailed document that provides complete information about the project.
- It includes an introduction, objectives, system design, implementation, results, and conclusions.

- The report should also describe technologies used, methodologies followed, and challenges faced.
- A well-structured report serves as official documentation and helps others understand the project easily.

Screenshots / API Documentation



Challenges & Solutions

➔ This section highlights the main difficulties encountered during the project and how they were solved.

➔ Common challenges include coding errors, integration issues, performance bottlenecks, and deployment failures.

➔ For each challenge, a proper explanation and implemented solution are described.

➔ This helps in reflecting the problem-solving and analytical skills demonstrated during the project.

GitHub README & Setup Guide

✓ The GitHub README file provides an overview of the project for developers and users.

✓ It typically includes the project description, features, technologies used, installation steps, and usage instructions.

✓ The setup guide explains how to clone the repository, install dependencies, and run the project locally.

✓ A clear and complete README ensures easy collaboration and understanding of the project.

Final Submission (Repo + Deployed Link)

- The final submission involves sharing the complete GitHub repository and the deployed project link.
- The repository should contain all source codes, documentation, and related assets properly organized.
- The deployed link allows evaluators to access the live version of the project online for testing.
- This marks the final step of the project, showcasing the functional and deployable version to stakeholders.

DEPLOYMENT LINK: https://product-catalog-vsc.netlify.app