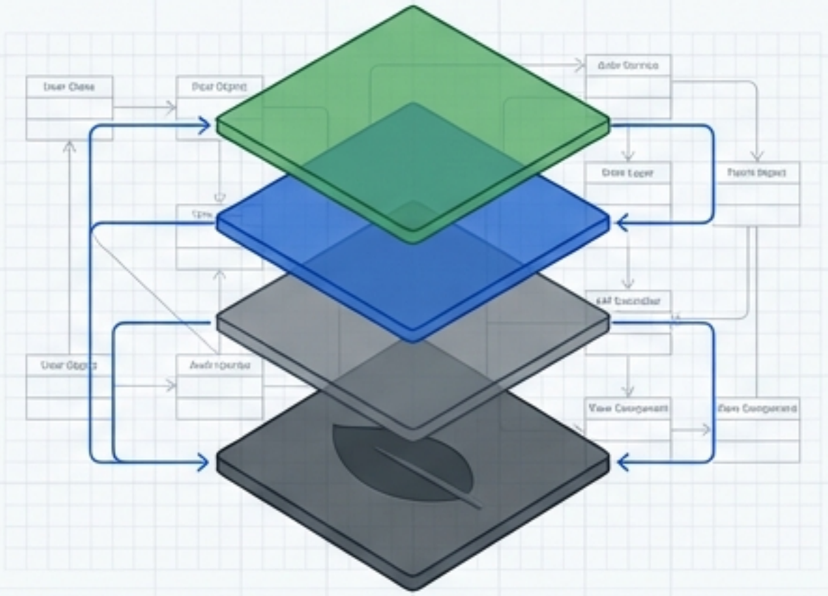
A Production-Ready MERN Application Built on OOP Principles

This presentation outlines a full-stack, enterprise-grade blogging platform designed to demonstrate a mastery of the MERN stack through the rigorous application of Object-Oriented Programming principles and modern web development best practices.



MongoDB | Express.js | React | Node.js | Secure | Scalable

Designed for Academic Excellence and Production Readiness

Academic & Architectural Highlights

- **Complete OOP Implementation:** Demonstrates all four pillars: Encapsulation, Abstraction, Inheritance, and Polymorphism.
- **Layered Architecture:** A clean separation of concerns from Routes through to Models.
- **100% MongoDB Persistence:** All application state is database-backed with zero reliance on frontend-only state.

Production-Ready Features

- **Robust Security:** JWT authentication, password hashing, and role-based password hashing, and role-based access control are built-in.
- **Scalable by Design:** The service-oriented architecture allows for future expansion and maintenance.
- **Comprehensive Error Handling & Validation:** Ensures system stability and a predictable user experience.

Secure User Identity & Access Management

User Registration & Hashing

- ✓ Secure user signup process.
- ✓ Passwords are never stored in plain text; they are hashed using `bcrypt` with a cost factor of 10 rounds.

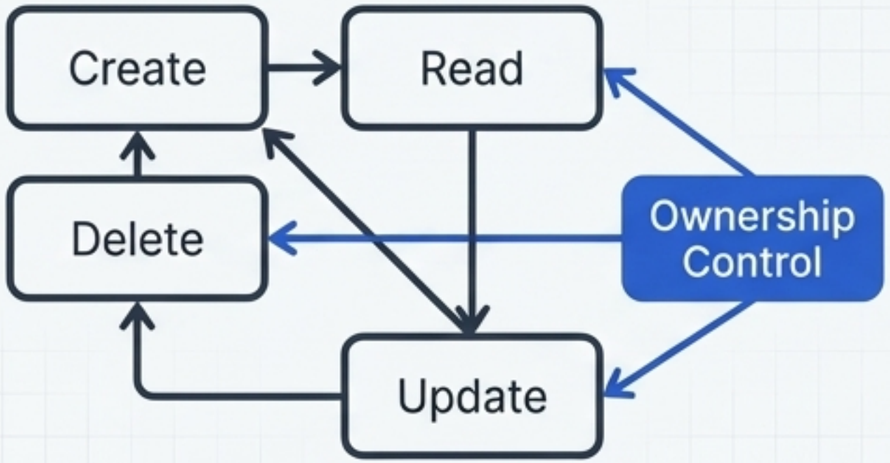
JWT-Based Authentication

- ✓ **Access Tokens**: Short-lived (15 minutes) for securing API requests.
- ✓ **Refresh Tokens**: Long-lived and stored securely in the database to maintain user sessions.

Session & Route Control

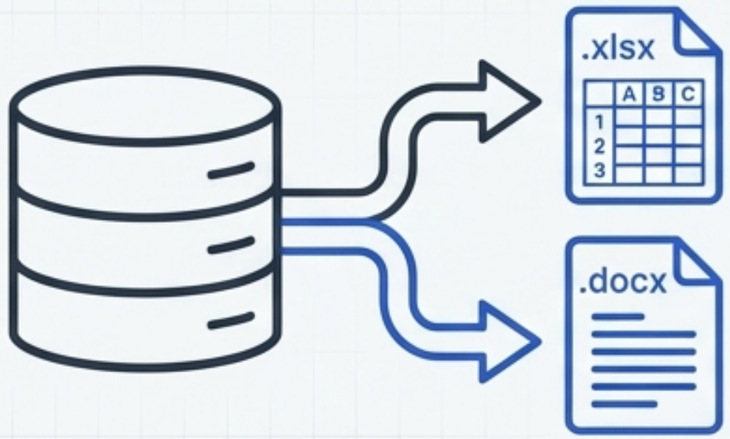
- ✓ **Session Persistence**: Seamless, automatic login for users on page refresh.
- ✓ **Protected Routes**: Role-based access control ensures that critical actions are restricted to authenticated and authorised users.
- ✓ **Security Best Practice**: An explicit login is required after signup; there is no automatic login on registration.

Robust Content Management and Professional Data Export



Complete Blog Management (CRUD)

- **Create:** Users can create posts with rich content, including titles, descriptions, categories, and images.
- **Read:** A dedicated, detailed view is available for individual posts.
- **Update:** Full editing functionality is available for existing posts.
- **Delete:** Posts can be permanently removed with a confirmation step.
- **Ownership Control:** A critical security feature ensures that only the original author of a post can edit or delete it.

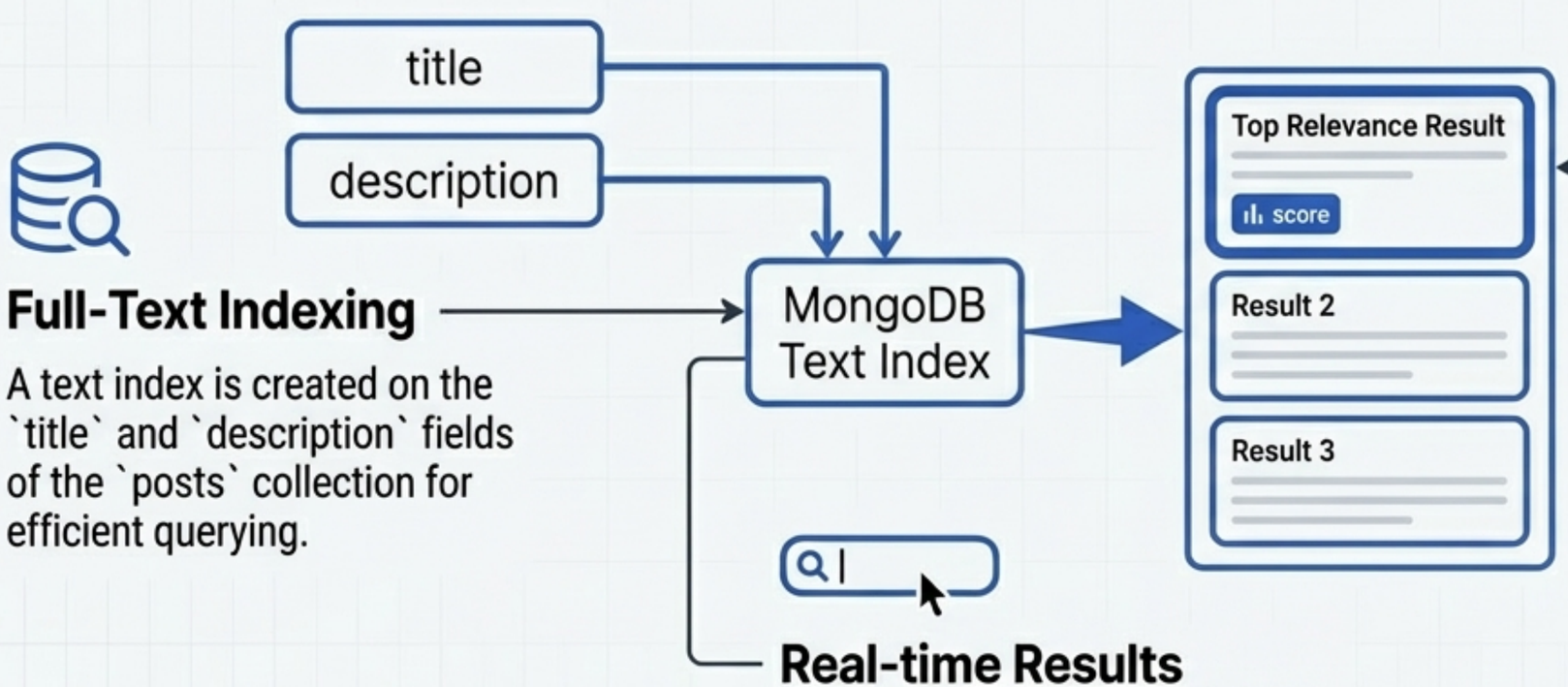


Professional File Export

- **Excel Export:** All blog posts can be exported to a formatted `.xlsx` file, including headers and table styling.
- **Word Export:** Individual blog posts can be exported as professionally formatted `.docx` documents.
- **Backend Handling:** Correct MIME types and download triggers are handled by the backend for a seamless user experience.

Intelligent Information Retrieval with Advanced Search

The search functionality is powered by MongoDB's native full-text search capabilities, ensuring high performance and relevance.



Full-Text Indexing

A text index is created on the `title` and `description` fields of the `posts` collection for efficient querying.

Relevance Scoring

Search results are automatically sorted by their relevance score, presenting the most pertinent information to the user first.

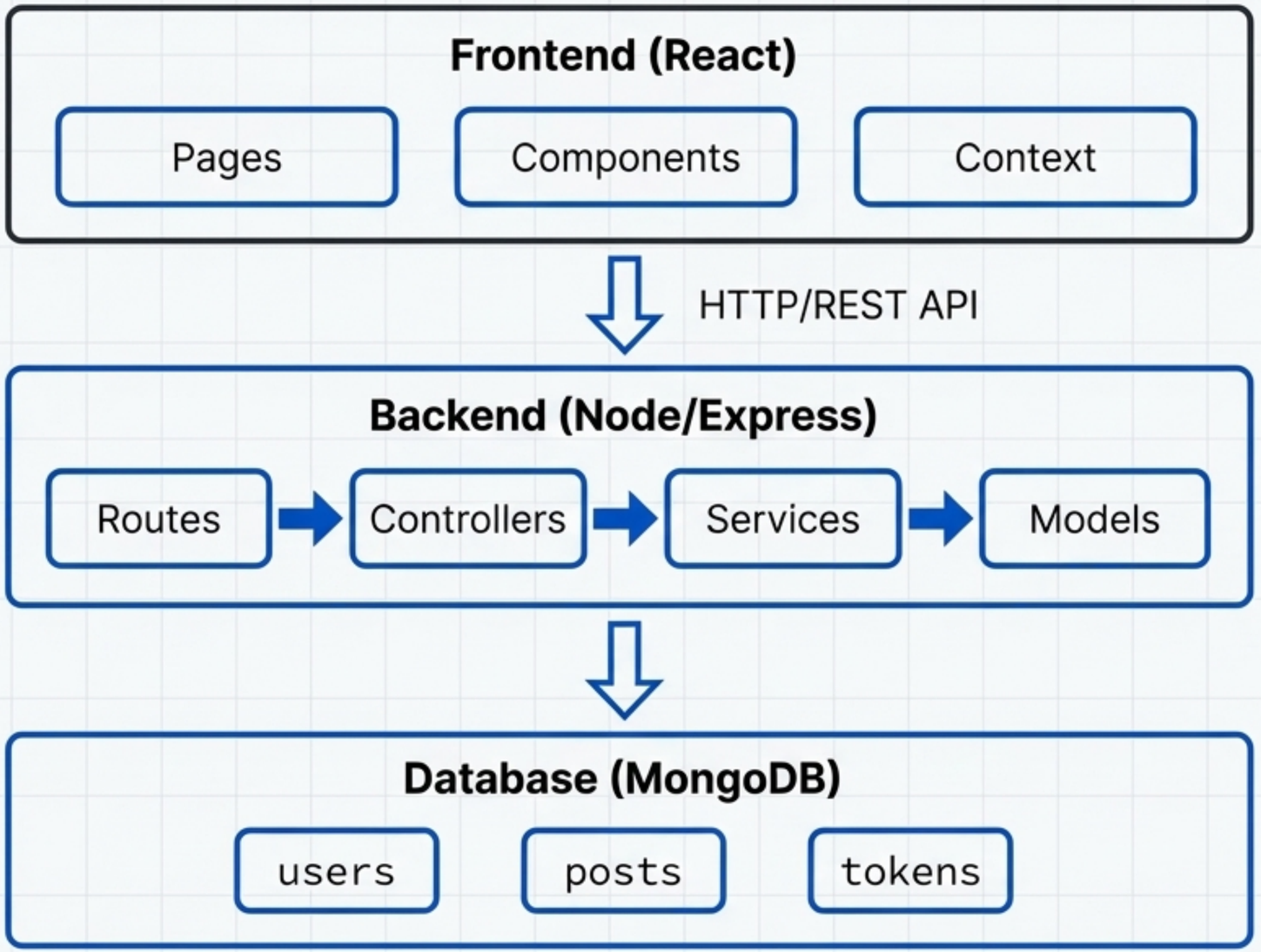
Real-time Results

The UI provides instant feedback, displaying results as the user types.

Fallback Support

In the event a text index is unavailable, the system gracefully degrades to a Regex-based search.

The System Blueprint: A Clean, Layered Architecture



🔑 This architecture ensures maintainability, scalability, and testability by isolating business logic from the transport layer and database implementation. Controllers are unaware of how data is stored, and the frontend is unaware of the backend's internal logic.

OOP in Practice: Encapsulation & Abstraction

Encapsulation

Bundling data and the methods that operate on that data into a single unit, hiding the internal state.

Project Evidence:

- **Models:** Mongoose schemas bundle data fields with their corresponding validation rules.
- **Services:** All business logic is contained within service classes, hiding implementation details from the controllers.
- **Private Methods:** Helper methods within services are kept private to the class interface.

```
class UserService extends BaseService {
  // Public interface
  async createUser(userData) { /* ... */ }

  // Private helper methods, hidden from controller
  _generateAccessToken(user) { /* ... */ }
  _sanitizeUser(user) { /* ... */ }
}
```

Abstraction

Hiding complex implementation details and exposing only the essential features of an object.

Project Evidence:

- **BaseService:** Provides an abstract interface for CRUD operations (create, findById), hiding the underlying MongoDB and Mongoose calls.
- **API Routes:** The frontend interacts with simple API endpoints, completely unaware of the complex business logic occurring on the backend.

```
class BaseService {
  async create(data) {
    /* Complex MongoDB logic is hidden here */
  }
  async findById(id) {
    /* Mongoose implementation is hidden here */
  }
}
```

OOP in Practice: Inheritance & Polymorphism

Inheritance

A mechanism where a new class derives properties and methods from an existing class, promoting code reuse.

Project Evidence

- UserService and PostService both extend a common BaseService.
- This allows them to inherit standard CRUD functionality without duplicating code. Each service is then free to add its own specialised methods.

Caption in Source Code Pro

```
class PostService extends BaseService {
  constructor() {
    super(Post); // Inherits base functionality for the Post model
  }
  // ... specialised post methods
}
```

Polymorphism

The ability of an object to take on many forms, often implemented via method overriding.

Project Evidence

- **Method Overriding**: The PostService overrides the base getAll method to implement custom logic for searching and filtering, providing a different behaviour for the same interface.
- **Format-Specific Exports**: exportToExcel() and exportToWord() provide different outputs through a similar conceptual interface.

Antation in Source Code Pro

```
// In PostService, this method overrides a more generic
// method from BaseService to add search logic.
async getAllPosts(username, category, search) {
  // Custom MongoDB text search implementation
}
```

The Technology Stack

Backend

- **Node.js (v18+):** JavaScript runtime environment.
- **Express.js (v4.18+):** Web application framework.
- **MongoDB (v6.0+):** NoSQL database for data persistence.
- **Mongoose (v8.0+):** Object Data Modeling (ODM) library for MongoDB.
- **jsonwebtoken (v9.0+):** For generating and verifying JWTs.
- **bcryptjs (v2.4+):** For password hashing.
- **exceljs / docx:** Libraries for professional file generation.
- **dotenv / cors / nodemon:** Essential development and utility libraries.

Frontend

- **React (v18.2+):** JavaScript library for building user interfaces.
- **Vite (v5.0+):** Modern frontend build tool and development server.
- **React Router DOM (v6.20+):** For client-side routing and navigation.
- **Axios (v1.6+):** Promise-based HTTP client for API communication.
- **Lucide React:** Icon library for a clean UI.

The API Contract: A RESTful Interface

Base URL: `http://localhost:8000/api`

Authentication

POST `/signup`
Register a new user.

POST `/login`
Authenticate a user and receive JWTs.

POST `/logout`
Invalidate a user's refresh token.

Blog Posts (CRUD)

GET `/posts`
Retrieve all posts with optional query params for search (`?search=`), category (`?category=`), and user (`?username=`).

GET `/post/:id`
Fetch a single post by its ID.

POST `/create`
Create a new post (Protected Route).

PUT `/update/:id`
Update an existing post (Protected Route, Ownership Verified).

DELETE `/delete/:id`
Remove a post (Protected Route, Ownership Verified).

File Export

GET `/posts/export/excel`
Triggers download of an Excel file with all posts.

GET `/posts/:id/export/word`
Triggers download of a Word document for a single post.

Note: Consistent JSON error responses are provided for status codes `400`, `401`, `403`, `404`, and `500`.

Ensuring System Integrity: Verification and Security

Testing & Verification

- **Manual Test Checklists**: Detailed checklists exist for critical user flows including Authentication (Signup, Login, Logout) and CRUD Operations (Create, Read, Update, Delete with persistence checks).
- **Automated Verification Scripts**:
  - `verify_mongodb.js`: A script to confirm database connectivity, data persistence, and the presence of required text search indexes.
  - `verify_users_db.js`: A script to validate the users collection.
- **Database Inspection**: Manual verification is supported via MongoDB Compass to directly inspect the `users`, `posts`, and `tokens` collections.

Key Security Features

- **Password Security**: `bcrypt` hashing (10 rounds) prevents plain-text password storage.
- **Authorization**: Backend ownership checks prevent unauthorised modification of data.
- **Input Validation**: Mongoose schemas enforce required fields and unique constraints (username, email, title).
- **CORS**: Correctly configured to allow requests only from the designated frontend application.
- **Centralised Error Handling**: A global middleware sanitises error messages in production to prevent data leakage.

From Local Setup to Production Deployment

Local Development Setup (Abridged)

- Prerequisites: Node.js (v18+), MongoDB (v6.0+).

Backend:

1. `cd backend`
2. `npm install`
3. Create and configure `.env` file with database credentials and secret keys.
4. `npm run dev`

Frontend:

1. `cd frontend`
2. `npm install`
3. `npm run dev`

Production Deployment Checklist

Backend (e.g., Railway, Render):

1. Configure production environment variables (`NODE_ENV=production`).
2. Use a cloud database connection string (e.g., MongoDB Atlas).
3. Generate secure, random strings for `ACCESS_SECRET_KEY` and `REFRESH_SECRET_KEY`.

Frontend (e.g., Vercel, Netlify):

1. Update the Axios base URL in `frontend/src/services/api.js` to point to the live backend URL.
2. `npm run build` to create a production-optimised build.
3. Deploy the generated `dist` folder.

Project Status: Complete, Verified, and Production Ready

- **Features Implemented:** All core features, including authentication, CRUD, **search**, and file export, are complete and fully functional.
- **Database Persistence Verified:** Automated scripts and manual checks confirm that all data is correctly and persistently stored in MongoDB.
- **OOP Principles Demonstrated:** The codebase serves as a practical, documented example of all four pillars of Object-Oriented Programming.
- **Security Measures in Place:** The application is built with security as a priority, implementing hashing, JWTs, and authorisation checks.
- **Professional Documentation Complete:** The project is supported by comprehensive documentation covering architecture, setup, and API usage.