Project Requirements Document (PRD) - "Inkwell" (Your Medium.com)

Project Name: Inkwell (Placeholder - feel free to rename!)
Version: 1.0
Date: July 9, 2025
Author: [Your Name/Team]

1. Introduction
Inkwell is a sophisticated web platform designed to empower individuals to publish, discover, and engage with high-quality written content. Inspired by the success of platforms like Medium.com, Inkwell aims to provide a seamless and intuitive experience for both content creators and consumers, fostering a vibrant community around shared knowledge and diverse perspectives. This document outlines the functional and non-functional requirements for the initial release of Inkwell.

2. Goals & Objectives
 * Primary Goal: To create a robust, scalable, and user-friendly content publishing and consumption platform.
 * Objective 1: Enable users to easily create, edit, publish, and manage their articles (posts).
 * Objective 2: Provide an intuitive content discovery experience through personalized feeds, topics, and search.
 * Objective 3: Foster community engagement through features like comments, claps (likes), and following.
 * Objective 4: Ensure high performance, security, and scalability to support a growing user base and content volume.
 * Objective 5: Implement a clean, modern, and responsive UI/UX across devices.

3. Target Audience
 * Aspiring and established writers/bloggers
 * Knowledge seekers and casual readers
 * Professionals looking to share insights
 * Communities interested in specific topics

4. Functional Requirements
4.1. User Management & Authentication
 * FR-UM-001: User Registration (Email/Password, Google/Facebook SSO)
 * FR-UM-002: User Login/Logout
 * FR-UM-003: Password Reset/Forgot Password functionality
 * FR-UM-004: User Profile Management (Name, Bio, Profile Picture, Social Links
 * FR-UM-005: Email Verification

4.2. Content Creation & Management (Author Features)
 * FR-CC-001: Rich Text Editor for post creation (WYSIWYG - bold, italic, headings, links, images, code blocks, lists)
 * FR-CC-002: Post Drafts: Auto-save and manual saving of drafts.
 * FR-CC-003: Publish/Unpublish Post functionality.
 * FR-CC-004: Edit/Delete Published Posts (with versioning/history optional for future).
 * FR-CC-005: Tagging/Categorization of posts.
 * FR-CC-006: Featured Image/Thumbnail for posts.
 * FR-CC-007: Post Metadata (Reading Time, Publish Date).
 * FR-CC-008: User Dashboard for managing all authored posts (drafts, published).

4.3. Content Discovery & Consumption (Reader Features)
 * FR-CD-001: Personalized Home Feed (based on followed users/topics).
 * FR-CD-002: Explore/Discover Page (trending posts, popular topics).
 * FR-CD-003: Search functionality (by keywords, authors, tags).
 * FR-CD-004: Individual Post View with comments section.
 * FR-CD-005: Clapping/Liking Posts.
 * FR-CD-006: Commenting on Posts (replying to comments).
 * FR-CD-007: Bookmarking/Saving Posts for later.
 * FR-CD-008: Following Authors/Topics.
 * FR-CD-009: Related Posts recommendations.

4.4. Notifications
 * FR-NF-001: In-app notifications for new followers, comments on user's posts, claps on user's posts.
 * FR-NF-002: Email notifications (optional, user configurable).

5. Non-Functional Requirements
 * Performance:
   * Load time for pages < 3 seconds.
   * API response times < 500ms for critical operations.
   * Capable of handling 1000 concurrent users without significant degradation.
 * Scalability:
   * System architecture designed to scale horizontally.
   * Database capable of handling growing data volumes.
 * Security:
   * Robust authentication and authorization mechanisms (JWT).
   * Protection against common web vulnerabilities (XSS, CSRF, SQL Injection).
   * Data encryption (in transit and at rest for sensitive data).
 * Usability:
   * Intuitive and consistent user interface.

  * Responsive design for various devices (desktop, tablet, mobile).
* Reliability:
  * High availability (e.g., 99.9% uptime).
  * Error logging and monitoring.
* Maintainability:
  * Modular and well-documented codebase.
  * Clear separation of concerns.

6. Out of Scope for v1.0
* Monetization (e.g., paid subscriptions, author earnings)
* Advanced analytics for authors
* Multi-language support
* Offline mode
* Admin dashboard for content moderation (basic moderation via reports could be added if critical)

UI/UX Design Specification - "Inkwell"
Note: This section will provide a detailed conceptual overview. Actual UI/UX design involves wireframes, mockups (Figma/Adobe XD), and interactive prototypes.
1. Design Principles
* Clean & Minimalist: Focus on content, reduce visual clutter.
* Content-First: Prioritize readability and content presentation.
* Intuitive Navigation: Easy access to core features.
* Responsive: Optimal experience across all devices.
* Engaging: Encourage interaction and community building.
* Consistent: Maintain a unified look and feel across the platform.
2. Branding & Visuals
* Logo: A simple, elegant logo representing writing/creativity.
* Color Palette:
  * Primary: A calm, professional color (e.g., deep blue, forest green) for brand elements.
  * Secondary: A complementary vibrant color for accents, CTAs (e.g., orange, teal).
  * Neutral: Grayscale for text, backgrounds, and subtle UI elements (various shades of grey for hierarchy).
* Typography:
  * Headings: A strong, readable serif or sans-serif font.
  * Body Text: A highly readable sans-serif font optimized for long-form reading (e.g., Georgia, Open Sans, Lato).
  * Line Height & Spacing: Optimized for readability.
* Imagery: High-quality, relevant images for post thumbnails and user profiles.
3. Key Screen Layouts & Elements
3.1. General Layout
* Header:

  * Logo (left)
  * Search Bar (middle/right)
  * Navigation (e.g., Home, Explore, Write, Notifications, User Avatar)
  * Login/Signup (if not logged in)
* Footer: Copyright, About Us, Privacy Policy, Terms of Service, Social Media links.
* Sidebar (Optional/Contextual): For specific pages like article view (e.g., author info, related posts).

3.2. Specific Screens

a) Landing Page (Guest User)
* Hero section with catchy headline and call to action (e.g., "Write your story," "Discover great ideas").
* Showcase of trending/popular articles.
* "Join Inkwell" / "Sign Up" prominent buttons.
* Brief explanation of benefits.

b) Home Feed (Logged-in User)
* Personalized stream of articles from followed authors/topics.
* Card-based layout for articles:
  * Featured Image
  * Title
  * Snippet/First few lines
  * Author Name, Date, Reading Time
  * Tags
  * Clap/Bookmark icons
* Infinite scrolling or pagination.

c) Explore Page
* Curated categories/topics.
* Trending articles.
* "Recommended for you" section.
* Search filter options.

d) Article View Page
* Hero Section: Featured Image, Title, Author Name, Publish Date, Reading Time.
* Author Bio: Small section with author's profile picture, name, "Follow" button.
* Content Area: Rich text formatted content, optimized for readability (sufficient line height, paragraph spacing).

* Engagement Section:
  * Clap button (with count)
  * Bookmark button
  * Share buttons (social media)

* Comments Section:
  * Input field for new comments.
  * List of comments, nested replies.
  * Ability to like comments.
* Related Articles Section: Below the comments.

e) Write/Edit Post Page
* Header: Title input, Publish/Save Draft buttons.
* Rich Text Editor: Prominent, full-width editor with toolbar (bold, italic, H1-H6, lists, links, images, code block, blockquote).
* Sidebar/Modal for Settings: Tags, Featured Image Upload, SEO settings (optional).
* Preview Mode: Ability to see how the post will look before publishing.

f) User Profile Page
* Profile Picture, Name, Bio.
* Follow/Unfollow button.
* List of authored posts.
* "Clapped Posts," "Saved Posts" sections (tabs).

g) Authentication Flows (Login/Register)
* Clean, focused modal or dedicated page.
* Clear input fields, password visibility toggle.
* Google/Facebook SSO buttons.
* "Forgot Password" link.

4. Interaction Design
* Micro-interactions: Visual feedback for claps, saves, button clicks.
* Transitions & Animations: Smooth, subtle animations for navigation and content loading.
* Form Validation: Real-time feedback for input errors.
* Notifications: Non-intrusive pop-ups or a dedicated notification center.

API Contract / Endpoint Documentation - "Inkwell"

Base URL: https://api.inkwell.com/api/v1 (or http://localhost:5000/api/v1 for development)
Authentication: JWT (Bearer Token in Authorization header)
Content-Type: application/json for requests and responses.
1. Authentication Endpoints
1.1. User Registration
* Endpoint: POST /auth/register
* Description: Registers a new user.
* Request Body:

```
{
 "username": "john_doe",
 "email": "john.doe@example.com",
```

```
    "password": "StrongPassword123!",
    "confirmPassword": "StrongPassword123!"
}
```

* Response (201 Created):

```
  {
    "message": "User registered successfully. Please verify your email.",
    "user": {
        "_id": "user_id_here",
        "username": "john_doe",
        "email": "john.doe@example.com"
    }
}
```

* Error Responses (400 Bad Request, 409 Conflict):

```
  { "error": "Email already registered" }
```

1.2. User Login

* Endpoint: POST /auth/login
* Description: Authenticates a user and returns a JWT token.
* Request Body:

```
  {
    "email": "john.doe@example.com",
    "password": "StrongPassword123!"
}
```

* Response (200 OK):

```
  {
    "token": "eyJhbGciOiJIUzI1Ni...",
    "user": {
        "_id": "user_id_here",
        "username": "john_doe",
        "email": "john.doe@example.com",
        "profilePicture": "url_to_image"
    }
}
```

1.3. Forgot Password Request

* Endpoint: POST /auth/forgot-password
* Description: Sends a password reset link to the user's email.

* Request Body:

```
  {
   "email": "john.doe@example.com"
}
```

 * Response (200 OK): {"message": "Password reset link sent to your email."}

1.4. Reset Password

 * Endpoint: PUT /auth/reset-password/:token
 * Description: Resets the user's password using a valid token.
 * Request Body:

```
  {
   "newPassword": "NewStrongPassword123!",
   "confirmNewPassword": "NewStrongPassword123!"
}
```

 * Response (200 OK): {"message": "Password reset successfully."}

2. User Endpoints

2.1. Get User Profile

 * Endpoint: GET /users/:userId
 * Description: Retrieves a user's public profile.
 * Response (200 OK):

```
  {
   "_id": "user_id_here",
   "username": "jane_doe",
   "email": "jane.doe@example.com",
   "bio": "Passionate writer and tech enthusiast.",
   "profilePicture": "url_to_image",
   "followersCount": 150,
   "followingCount": 75,
   "createdAt": "2023-01-15T10:00:00Z"
}
```

2.2. Update User Profile (Authenticated)

 * Endpoint: PUT /users/profile
 * Description: Updates the authenticated user's profile.
 * Request Body: (Partial update allowed)

```
  {
   "username": "jane_updated",
   "bio": "Updated bio.",
   "profilePicture": "new_url_to_image"
}
```

 * Response (200 OK): Updated user object.

2.3. Follow User (Authenticated)
 * Endpoint: POST /users/:userId/follow
 * Description: Follows a specific user.
 * Response (200 OK): {"message": "Successfully followed user."}

2.4. Unfollow User (Authenticated)
 * Endpoint: POST /users/:userId/unfollow
 * Description: Unfollows a specific user.
 * Response (200 OK): {"message": "Successfully unfollowed user."}

3. Post Endpoints
3.1. Create New Post (Authenticated)
 * Endpoint: POST /posts
 * Description: Creates a new article post.
 * Request Body:

```
  {
   "title": "My First MERN Stack App",
   "content": "<h1>Introduction</h1><p>This is the content of my post...</p>",
   "tags": ["MERN", "WebDev", "Beginner"],
   "featuredImage": "url_to_featured_image.jpg",
   "isDraft": false // true if saving as draft
}
```

 * Response (201 Created): Newly created post object.

3.2. Get All Posts (Paginated, Filterable)
 * Endpoint: GET /posts?page=1&limit=10&tag=MERN&authorId=abc
 * Description: Retrieves a list of posts. Supports pagination, filtering by tags, and author.
 * Response (200 OK):

```
  {
   "posts": [
      {
         "_id": "post_id_1",
         "title": "Post Title 1",
         "author": { "_id": "author_id", "username": "author_name" },
         "featuredImage": "url",
         "readingTime": "5 min read",
         "clapsCount": 120,
         "commentsCount": 15,
         "createdAt": "2023-07-01T10:00:00Z"
      }
      // ... more posts
   ],
```

```
    "currentPage": 1,
    "totalPages": 5,
    "totalPosts": 50
}
```


3.3. Get Single Post by ID
 * Endpoint: GET /posts/:postId
 * Description: Retrieves a single post with its full content and comments.
 * Response (200 OK):


```
  {
    "_id": "post_id_1",
    "title": "Post Title 1",
    "content": "Full HTML content of the post...",
    "author": { "_id": "author_id", "username": "author_name", "profilePicture": "url" },
    "tags": ["MERN", "WebDev"],
    "featuredImage": "url",
    "readingTime": "5 min read",
    "clapsCount": 120,
    "commentsCount": 15,
    "createdAt": "2023-07-01T10:00:00Z",
    "updatedAt": "2023-07-01T11:30:00Z",
    "comments": [
        {
            "_id": "comment_id_1",
            "user": { "_id": "user_id_a", "username": "user_a" },
            "content": "Great article!",
            "createdAt": "2023-07-02T12:00:00Z",
            "replies": []
        }
    ]
}
```


3.4. Update Post (Authenticated, Author Only)
 * Endpoint: PUT /posts/:postId
 * Description: Updates an existing post.
 * Request Body: (Partial update allowed)


```
  {
    "title": "Updated Title",
    "content": "New updated content.",
    "isDraft": false
}
```


 * Response (200 OK): Updated post object.
3.5. Delete Post (Authenticated, Author Only)

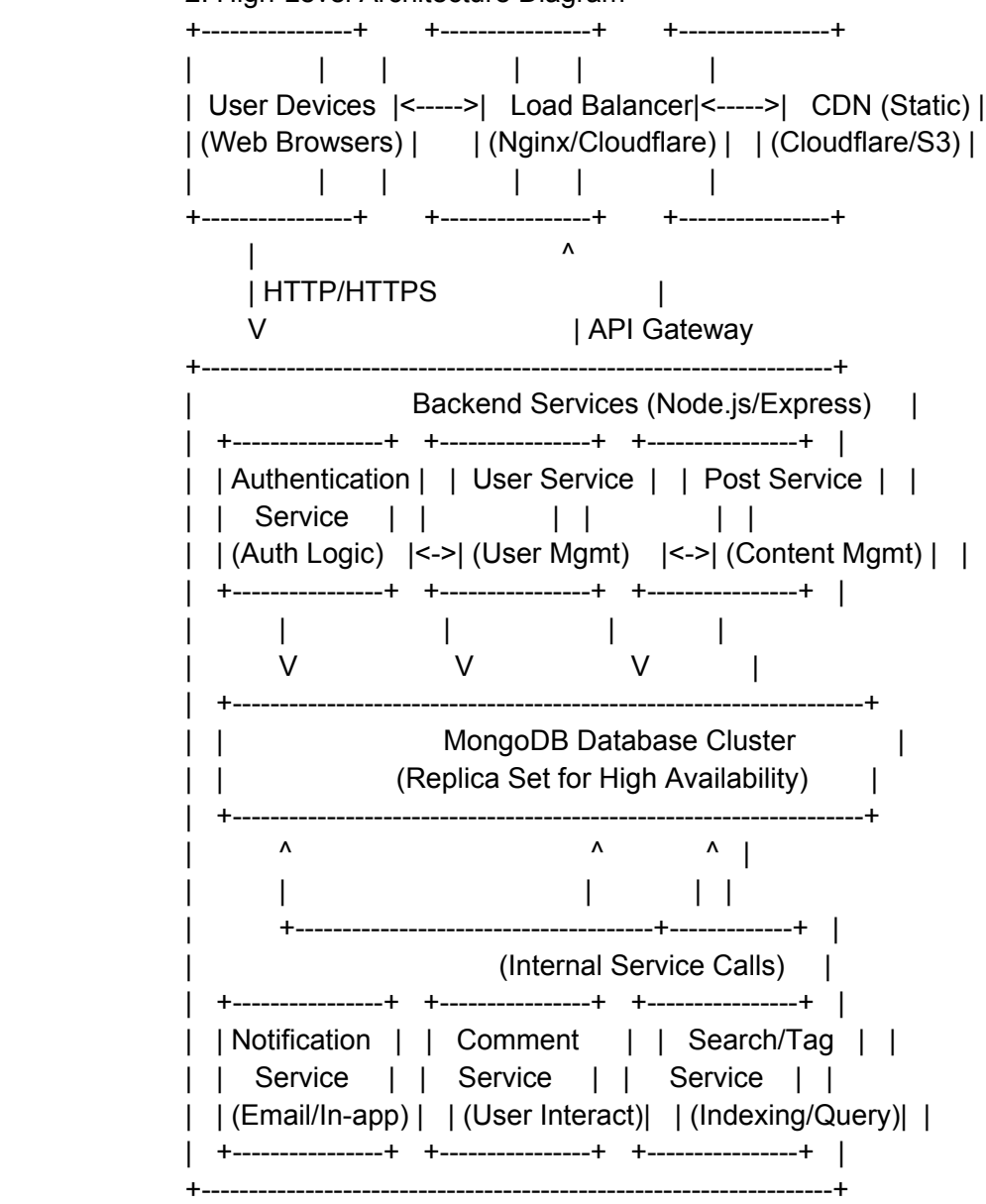
 * Endpoint: DELETE /posts/:postId
 * Description: Deletes a post.
 * Response (204 No Content): (Empty response)
4. Comment Endpoints
4.1. Add Comment to Post (Authenticated)
 * Endpoint: POST /posts/:postId/comments
 * Description: Adds a new comment to a post.
 * Request Body:
   {
    "content": "This is a new comment.",
    "parentId": "optional_comment_id_for_reply"
}

 * Response (201 Created): New comment object.
4.2. Delete Comment (Authenticated, Author or Post Author)
 * Endpoint: DELETE /comments/:commentId
 * Description: Deletes a comment.
 * Response (204 No Content): (Empty response)
5. Engagement Endpoints
5.1. Clap/Like a Post (Authenticated)
 * Endpoint: POST /posts/:postId/clap
 * Description: Toggles a clap/like on a post.
 * Response (200 OK): {"message": "Clap toggled successfully.", "newClapCount": 121}
5.2. Bookmark/Save a Post (Authenticated)
 * Endpoint: POST /posts/:postId/bookmark
 * Description: Toggles saving a post for later.
 * Response (200 OK): {"message": "Bookmark toggled successfully.", "isBookmarked": true}
6. Search & Tags Endpoints
6.1. Search Posts
 * Endpoint: GET /search?q=keyword&type=post
 * Description: Searches for posts by title or content keywords.
 * Response (200 OK): List of matching posts.
6.2. Get Popular Tags
 * Endpoint: GET /tags/popular
 * Description: Retrieves a list of popular tags.
 * Response (200 OK):
   [
    {"name": "MERN", "count": 150},
    {"name": "React", "count": 120}
]

System Architecture Document - "Inkwell"
1. Overview
Inkwell will be built using a microservices-oriented approach (though initially, some services

might be co-located or monolith-lite for faster iteration) following the MERN (MongoDB, Express.js, React.js, Node.js) stack. This architecture promotes scalability, maintainability, and independent deployment of components.

2. High-Level Architecture Diagram



3. Component Breakdown

3.1. Frontend (React.js)

* Technology: React.js, React Router, Redux/React Context for state management, Axios for API calls, Styled Components/Tailwind CSS for styling.
* Deployment: Static files served via CDN (e.g., Cloudflare, AWS S3/CloudFront).
* Key Responsibilities:

  * User Interface rendering.
  * User interaction handling.
  * Client-side routing.
  * State management.
  * API request orchestration.

3.2. Backend (Node.js/Express.js)

* Technology: Node.js, Express.js, Mongoose (ODM for MongoDB), JWT for authentication, bcrypt for password hashing, Nodemailer for email.
* Deployment: Containerized (Docker) and orchestrated (Kubernetes/ECS) for production; directly on VM/server for initial development.
* Core Services (Logical Separation - can be separate microservices or one monolith-lite for v1):
  * Authentication Service:
    * User registration, login, logout.
    * Password reset/forgot password.
    * JWT token generation and validation.
  * User Service:
    * User profile management (CRUD).
    * Follow/Unfollow logic.
  * Post Service:
    * CRUD operations for articles/posts.
    * Handles rich text content storage.
    * Manages tags and featured images.
  * Comment Service:
    * CRUD operations for comments.
    * Handles nested comments (replies).
  * Engagement Service:
    * Manages claps/likes.
    * Manages bookmarks/saved posts.
  * Notification Service:
    * Handles in-app notifications.
    * Manages sending email notifications (e.g., for password resets, new followers).
  * Search/Tag Service:
    * Provides search capabilities.
    * Manages popular tags. (Could use a dedicated search engine like Elasticsearch later).

3.3. Database (MongoDB)

* Technology: MongoDB (NoSQL Document Database).
* Deployment: MongoDB Atlas (managed service) for production, or a self-hosted replica set for high availability and redundancy.
* Key Features Used:
  * Document storage for flexible schema.
  * Indexing for efficient queries.
  * Aggregation framework for complex data retrieval (e.g., trending posts).
  * Replica sets for fault tolerance and read scaling.

3.4. Other Components

* Load Balancer/API Gateway: (e.g., Nginx, AWS ALB, Kong)
  * Distributes incoming traffic across backend instances.
  * Handles SSL termination.
  * Provides a single entry point for API consumers.
  * Rate limiting, security policies (future).
* Content Delivery Network (CDN): (e.g., Cloudflare, AWS CloudFront)
  * Serves static assets (JS, CSS, images) from edge locations.
  * Reduces latency and improves load times for users globally.
  * Enhances security (DDoS protection).
* Email Service: (e.g., SendGrid, Nodemailer with SMTP)
  * For sending transactional emails (email verification, password resets, notifications).
* Storage for User Uploads: (e.g., AWS S3, Cloudinary)
  * For storing user profile pictures and featured images for posts.

4. Data Flow

* User Request: User interacts with the React frontend in their browser.
* Frontend API Call: React app makes an API request to the backend via the Load Balancer/API Gateway.
* Authentication/Authorization: API Gateway (or first middleware in Express) validates the JWT token.
* Backend Processing: The relevant Node.js service (e.g., Post Service) processes the request.
* Database Interaction: The service interacts with MongoDB to read or write data.
* Response: MongoDB returns data to the service, which processes it and sends a JSON response back through the API Gateway.
* Frontend Rendering: React app receives the JSON response and updates the UI.

5. Scalability Considerations

* Stateless Backend: All backend services will be stateless to allow for easy horizontal scaling.
* Database Scaling: MongoDB replica sets for read scaling, sharding for extreme data growth (future consideration).
* Caching: Redis for caching frequently accessed data (e.g., popular posts, user profiles) to reduce database load.
* Asynchronous Processing: Message queues (e.g., RabbitMQ, Kafka) for non-critical tasks like sending notifications, processing image uploads (future consideration).

6. Security Considerations

* JWT: Securely generated and validated tokens.
* Password Hashing: bcrypt for strong, one-way hashing.
* Input Validation: Strict validation on all incoming API requests to prevent injection attacks.
* CORS: Properly configured Cross-Origin Resource Sharing.
* HTTPS: All communication via HTTPS.
* Environment Variables: Sensitive information stored as environment variables, not hardcoded.
* Rate Limiting: Protect against brute-force attacks.

Database Schema / ER-Diagrams - "Inkwell" (MongoDB Document Structure)

MongoDB is a NoSQL document database, so we don't have traditional relational tables. Instead, we have collections, and documents within those collections. Relationships are typically

established through embedding (for small, frequently accessed, and tightly coupled data) or referencing (for larger, less frequently accessed, or loosely coupled data).

1. Key Principles for MongoDB Schema Design

* Embedding vs. Referencing:
  * Embedding: Use when data is highly related, rarely accessed independently, and grows together. Reduces queries. E.g., comments within a post document if comments are few and not heavily queried on their own.
  * Referencing: Use when data is large, frequently accessed independently, or has a many-to-many relationship. E.g., users referencing posts.
* Denormalization: Duplicate some data to avoid joins, which MongoDB doesn't natively support in the same way as SQL. E.g., storing authorName directly in Post document to avoid looking up User for every post.
* Indexes: Create indexes on fields used in queries (e.g., _id, email, createdAt, tags).

2. Collections and Document Structures

2.1. users Collection

* Purpose: Stores user profiles and authentication information.
* Fields:
  * _id: ObjectId (Primary Key)
  * username: String (Unique, Indexed)
  * email: String (Unique, Indexed)
  * password: String (Hashed)
  * profilePicture: String (URL to image)
  * bio: String
  * socialLinks: Object (e.g., { twitter: "...", linkedin: "..." })
  * followers: Array of ObjectId (References users._id) - This could be a separate follows collection for very large scale to avoid huge arrays.
  * following: Array of ObjectId (References users._id) - Similarly, could be a separate follows collection.
  * clappedPosts: Array of ObjectId (References posts._id) - Can be a large array, consider separate claps collection for very high volumes.
  * bookmarkedPosts: Array of ObjectId (References posts._id) - Can be a large array, consider separate bookmarks collection.
  * createdAt: Date
  * updatedAt: Date
  * isVerified: Boolean (for email verification)
  * resetPasswordToken: String (Temporary for password reset)
  * resetPasswordExpires: Date (Expiration for token)

2.2. posts Collection

* Purpose: Stores article content and metadata.
* Fields:
  * _id: ObjectId (Primary Key)
  * author: ObjectId (References users._id, Indexed)
  * authorInfo: Object (Denormalized: _id, username, profilePicture) - for quick display without extra lookup

* title: String (Indexed for search)
* content: String (HTML string of the rich text content)
* featuredImage: String (URL to image)
* tags: Array of String (Indexed for search/filtering)
* readingTime: String (e.g., "5 min read")
* clapsCount: Number (Default 0, Incremented/Decremented)
* commentsCount: Number (Default 0, Incremented/Decremented)
* isDraft: Boolean (Default false)
* status: String (e.g., 'published', 'draft', 'archived')
* createdAt: Date
* updatedAt: Date

2.3. comments Collection

* Purpose: Stores comments on posts. Designed for potential nesting.
* Fields:
  * _id: ObjectId (Primary Key)
  * post: ObjectId (References posts._id, Indexed)
  * user: ObjectId (References users._id)
  * userInfo: Object (Denormalized: _id, username, profilePicture)
  * content: String
  * parentId: ObjectId (Optional, References comments._id for replies, Indexed)
  * likesCount: Number (Default 0)
  * createdAt: Date
  * updatedAt: Date

2.4. notifications Collection (Optional for v1, could be simpler Activity log)

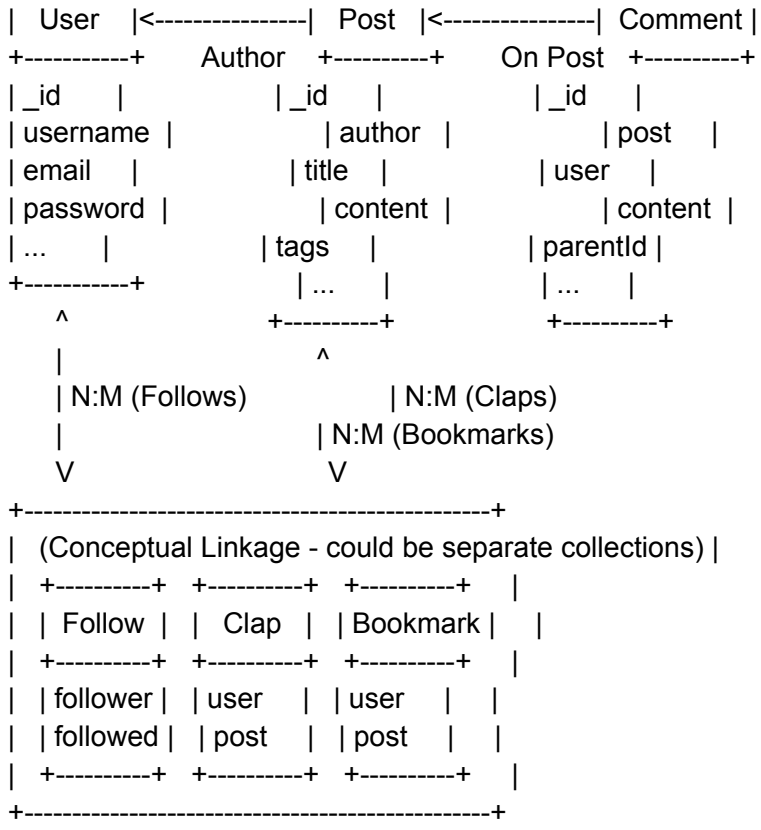
* Purpose: Stores in-app notifications for users.
* Fields:
  * _id: ObjectId
  * recipient: ObjectId (References users._id, Indexed)
  * type: String (e.g., 'new_follower', 'new_comment', 'post_clapped')
  * sourceUser: ObjectId (Optional, References users._id for who triggered it)
  * sourcePost: ObjectId (Optional, References posts._id)
  * message: String
  * isRead: Boolean (Default false)
  * createdAt: Date

2.5. tags Collection (Optional for trending tags, could be aggregated from posts)

* Purpose: To easily query and manage popular tags. Can be populated by a background process.
* Fields:
  * _id: ObjectId
  * name: String (Unique, Indexed)
  * count: Number (How many posts use this tag)
  * lastUsedAt: Date

3. ER-Diagram (Conceptual, focusing on relationships)

+----------+ 1:N +----------+ 1:N +----------+



Explanation of Relationships:

* User - Post (1:N): One user can author many posts. posts.author references users._id.
* Post - Comment (1:N): One post can have many comments. comments.post references posts._id.
* User - Comment (1:N): One user can make many comments. comments.user references users._id.
* Comment - Comment (1:N, Self-referencing): A comment can have many replies. comments.parentId references comments._id.
* User - User (N:M - Follows): A user can follow many users, and be followed by many users. This is typically implemented with a follows collection or by embedding arrays of _ids in User documents.
* User - Post (N:M - Claps): A user can clap for many posts, and a post can be clapped by many users. This is typically implemented with a claps collection or by embedding arrays of _ids.
* User - Post (N:M - Bookmarks): A user can bookmark many posts, and a post can be bookmarked by many users. Similar to claps, often a bookmarks collection or embedded arrays.

Note on Arrays: For followers, following, clappedPosts, bookmarkedPosts fields within the users document, if the number of entries becomes very large (e.g., thousands), it's generally better practice to create separate collections (e.g., follows, claps, bookmarks) to avoid performance issues with large documents and array updates. For an MNC-level project, especially with expected high user engagement, this would be a strong consideration. For v1, embedding might

be acceptable for simplicity if initial scale isn't massive.
Frontend Plan - "Inkwell" (React.js)
1. Project Setup & Core Libraries
 * Create React App / Vite: npx create-react-app inkwell-frontend or npm create vite@latest inkwell-frontend -- --template react
 * Routing: react-router-dom
 * State Management: redux-toolkit for global state (e.g., user auth, notifications) or React Context API for simpler shared state.
 * API Calls: axios
 * Styling: tailwind-css (highly recommended for rapid development and maintainability) or styled-components / emotion (for component-based styling).
 * Rich Text Editor: react-quill or Draft.js / Slate.js (more complex but powerful).
 * Form Handling: react-hook-form (efficient and flexible) or Formik.
 * Validation: yup (schema validation with react-hook-form).
 * Icons: react-icons (Font Awesome, Material Design, etc.)
 * Date Formatting: date-fns or moment.js (though date-fns is lighter).
2. Folder Structure (Example)
src/
├── assets/ // Images, fonts, etc.
├── components/ // Reusable UI components (buttons, cards, modals)
│ ├── Auth/
│ │ ├── LoginForm.jsx
│ │ └── RegisterForm.jsx
│ ├── Layout/
│ │ ├── Header.jsx
│ │ ├── Footer.jsx
│ │ └── MainLayout.jsx
│ ├── Post/
│ │ ├── PostCard.jsx
│ │ ├── PostContent.jsx
│ │ └── CommentSection.jsx
│ ├── UI/
│ │ ├── Button.jsx
│ │ ├── Input.jsx
│ │ └── Spinner.jsx
│ └── RichTextEditor.jsx
├── config/ // Environment variables, API constants
├── hooks/ // Custom React hooks
├── pages/ // Top-level page components (route-specific)
│ ├── Auth/
│ │ ├── LoginPage.jsx
│ │ ├── RegisterPage.jsx
│ │ └── ForgotPasswordPage.jsx
│ ├── HomePage.jsx

```
│   ├── ExplorePage.jsx
│   ├── PostDetailPage.jsx
│   ├── CreatePostPage.jsx
│   ├── EditPostPage.jsx
│   ├── ProfilePage.jsx
│   ├── NotFoundPage.jsx
│   └── SettingsPage.jsx
├── services/              // API service calls (axios instances, common methods)
│   ├── authService.js
│   ├── userService.js
│   ├── postService.js
│   └── commentService.js
├── store/                 // Redux/Context store setup
│   ├── index.js
│   ├── slices/            // Redux slices (authSlice, postSlice, userSlice)
│   └── reducers.js
├── styles/                // Global styles, Tailwind config
├── utils/                 // Utility functions (helpers, formatters)
├── App.jsx                // Main application component
├── index.css              // Base CSS
└── main.jsx               // Entry point
```

3. Core Features Implementation Plan

3.1. Authentication Flow

* Context/Redux Setup: Create authSlice (if Redux) or AuthContext to manage currentUser, isAuthenticated, loading, error.
* Login/Register Pages:
  * Forms using react-hook-form and yup for validation.
  * Call authService.login/register API.
  * Store JWT token in localStorage upon successful login.
  * Redirect to Home/Dashboard.
* Protected Routes: Use react-router-dom's Outlet and a wrapper component (ProtectedRoute) to check isAuthenticated status. Redirect to login if not authenticated.
* Logout: Clear token from localStorage and update state.

3.2. Post Creation & Editing

* CreatePostPage.jsx:
  * Integrate react-quill for rich text editing.
  * Form fields for Title, Tags, Featured Image Upload (handle client-side image preview and server-side upload).
  * react-hook-form for form management.
  * Draft/Publish buttons.
  * Call postService.createPost.
* EditPostPage.jsx:
  * Fetch post data by ID on component mount.

    * Populate editor and fields with existing data.
    * Update logic using postService.updatePost.
3.3. Content Display & Discovery
 * HomePage.jsx:
    * Fetch personalized feed using postService.getPosts with filters/parameters for followed authors/topics.
    * Map through posts and render PostCard components.
    * Implement infinite scrolling (Intersection Observer API or a library like react-infinite-scroll-component).
 * ExplorePage.jsx:
    * Fetch trending posts, popular topics/tags.
    * Search bar component.
 * PostDetailPage.jsx:
    * Fetch single post data using postService.getPostById.
    * Render PostContent component.
    * Render CommentSection component:
      * Display existing comments (nested).
      * Input field for new comments.
      * Call commentService.addComment.
 * ProfilePage.jsx:
    * Fetch user data and their authored posts.
    * Display profile details.
    * Follow/Unfollow button (call userService.follow/unfollow).
3.4. Engagement Features
 * Clapping:
    * Button on PostCard and PostDetailPage.
    * Optimistic UI update (increment count immediately) then send API request (engagementService.clapPost). Revert on error.
 * Bookmarking:
    * Button on PostCard and PostDetailPage.
    * Call engagementService.bookmarkPost.
 * Commenting:
    * Implemented within CommentSection of PostDetailPage.
    * Real-time updates for new comments (optional, could use polling or WebSockets for advanced).
3.5. Global Components
 * Header: Navigation links, search bar, user avatar/login buttons.
 * Footer: Standard links.
 * Notifications: Dedicated bell icon in header, pop-up for new notifications or dedicated page.
 * Loading Spinners/Skeletons: For better UX during API calls.
 * Error Boundaries: Catch unexpected UI errors.
4. Development Workflow
 * Component-Driven Development: Build small, reusable components first.
 * Mock APIs/Data: Use tools like JSON Server or directly mock axios calls for frontend

development while backend is in progress.
 * Linting & Formatting: ESLint, Prettier for code consistency.
 * Version Control: Git with feature branches, pull requests.
Authentication & Security Design Document - "Inkwell"
1. Overview
This document outlines the authentication and security measures for the Inkwell platform, focusing on protecting user data, ensuring data integrity, and securing access to resources. We will leverage JSON Web Tokens (JWT) for stateless authentication.
2. Core Security Principles
 * Least Privilege: Users/systems only have access to what they absolutely need.
 * Defense in Depth: Multiple layers of security controls.
 * Secure by Design: Security considered from the outset, not as an afterthought.
 * Fail Securely: Applications should fail in a way that does not expose sensitive information.
3. Authentication Mechanism (JWT)
3.1. Registration
 * User provides username, email, password, confirmPassword.
 * Backend validates input (format, strength, unique email).
 * Password is hashed using bcrypt (or similar strong, adaptive hashing algorithm) with a sufficient salt rounds (e.g., 10-12). Never store plain text passwords.
 * User record created in MongoDB.
 * (Optional but Recommended for MNC-level): Send email verification link. User account isVerified status is false until link clicked.
3.2. Login
 * User provides email and password.
 * Backend retrieves user by email.
 * Backend compares provided password with stored hashed password using bcrypt.compare().
 * If credentials match, a JSON Web Token (JWT) is generated.
   * Payload: Contains user ID (_id), username, and any roles (e.g., user, admin).
   * Secret: Signed with a strong, secret key stored as an environment variable (process.env.JWT_SECRET).
   * Expiration: Set a reasonable expiration time (e.g., 1 hour, 24 hours).
 * JWT is sent to the client (Frontend) in the response body.
 * Frontend stores the JWT securely, preferably in localStorage for ease of use (with CSRF considerations for httpOnly cookies). For an MNC project, httpOnly cookies are often preferred for security, but they introduce CSRF complexity for SPAs. For simplicity, localStorage is common in MERN tutorials, but know the trade-offs.
3.3. Subsequent Requests (Authorization)
 * Frontend includes the JWT in the Authorization header of every subsequent request (as Bearer <token>).
 * Backend middleware intercepts the request:
   * Verifies the JWT using the secret key.
   * Checks token expiration.
   * Decodes the payload to extract user information (_id, roles).
   * Attaches the user object to the req object (e.g., req.user = decodedPayload).

* Route handlers then access req.user to determine if the user is authenticated and authorized for the specific action (e.g., req.user._id must match postId.author for editing).

3.4. Token Refresh (Optional but Good Practice)

* To avoid frequent re-logins for users while keeping JWTs short-lived, implement refresh tokens.
* When the access token expires, the client sends a separate request with a long-lived refresh token (stored securely in httpOnly cookie).
* Server verifies the refresh token and issues a new access token.

4. Authorization

* Role-Based Access Control (RBAC):
  * Initial roles: user (default), admin (future: for moderation).
  * Middleware for routes: isAdmin, isAuthor (checks req.user._id against resource.authorId).
  * Example: router.put('/posts/:id', protect, authorize('author'), postController.updatePost);

5. Data Protection

5.1. Data in Transit

* HTTPS/SSL/TLS: All communication between client and server must use HTTPS. This encrypts data in transit, preventing eavesdropping and man-in-the-middle attacks.

5.2. Data at Rest

* Password Hashing: As mentioned, bcrypt is crucial.
* Sensitive Data Encryption: While MongoDB typically stores data unencrypted at rest by default (unless specifically configured at the filesystem/disk level), sensitive data fields (if any beyond hashed passwords, e.g., credit card info - not applicable for v1) might require application-level encryption before storage. MongoDB Atlas offers encryption at rest.

6. Vulnerability Prevention

* Cross-Site Scripting (XSS):
  * Frontend: Sanitize all user-generated content before rendering it on the page (e.g., using DOMPurify for rich text content).
  * Backend: Ensure robust input validation and sanitize data before saving to DB.
* Cross-Site Request Forgery (CSRF):
  * If using httpOnly cookies for JWTs/refresh tokens, implement CSRF tokens.
  * If primarily using localStorage for JWTs, CSRF is less of an immediate concern for that specific attack vector, but general HTTP methods protection still applies.
* SQL Injection (or NoSQL Injection):
  * Use Mongoose's ODM features (e.g., query builders, parameterization) which inherently protect against most NoSQL injection attempts by sanitizing inputs.
  * Never concatenate user input directly into database queries.
* Brute Force Attacks:
  * Rate Limiting: Implement rate limiting on login attempts, password reset requests, and potentially other API endpoints (e.g., express-rate-limit).
  * Account Lockout: Temporarily lock accounts after a certain number of failed login attempts.
* Broken Authentication & Session Management:
  * Secure JWT generation and validation.
  * Proper token expiration and invalidation on logout.
  * Avoid exposing sensitive information in JWT payload.

* Insecure Direct Object References (IDOR):
  * Always verify ownership of a resource before allowing an authenticated user to perform an action (e.g., ensure req.user._id matches the author of a post before allowing PUT or DELETE).
* Sensitive Data Exposure:
  * Do not return hashed passwords or other sensitive user data (like internal API keys) in API responses.
  * Proper error handling to avoid exposing internal server details in error messages.
* Parameter Tampering:
  * Validate all query parameters, request body data, and headers on the server-side.
  * Do not trust client-side input.

7. Logging & Monitoring
* Centralized Logging: Implement a logging system (e.g., Winston, Morgan for Express) to log security-relevant events (failed login attempts, unauthorized access attempts, critical errors).
* Monitoring & Alerting: Set up monitoring tools (e.g., Prometheus, Grafana, ELK stack) to detect suspicious activities and alert administrators.
    1.

8. Environment Variables
* All sensitive configurations (database URI, JWT secret, API keys for external services) must be stored as environment variables and not committed to version control.
* Use libraries like dotenv for local development.

This comprehensive breakdown should provide you with a solid foundation for building your MNC-level "Inkwell" application. Remember, this is a detailed blueprint, and the actual implementation will involve significant coding, testing, and iteration. Good luck on your journey to becoming a pro dev!