Designing a Wikipedia-type site where users can search for topics, edit text, and update the content requires careful consideration of both the technical and functional aspects. Here's a high-level design overview of such a system:

**1. Core Features**

* **Search Functionality**: Users should be able to search for articles by topic.
* **Edit Functionality**: Users can edit articles, track changes, and update the content.
* **Version Control**: Keep track of changes made by different users and allow reverting to previous versions.
* **User Management**: Authentication and authorization to control who can edit or view articles.
* **Moderation**: Admin users can review, approve, or revert edits if needed.

**2. System Architecture**

**Frontend (Client-Side):**

* **Technologies**: Use **React**, **Vue.js**, or **Angular** for building a dynamic, user-friendly interface.
* **Features**:
  + **Search Bar**: Allow users to search for articles by keyword.
  + **Article Display**: Render articles dynamically from the database.
  + **Editing Interface**: Provide a rich text editor (like **Quill**, **TinyMCE**, or **CKEditor**) that allows users to edit the article.
  + **Revision History UI**: Display the edit history with the option to compare versions and revert to a previous one.

**Backend (Server-Side):**

* **Technologies**: Use a backend framework like **Node.js**, **Django** (Python), or **Spring Boot** (Java) to handle server logic.
* **Features**:
  + **API Endpoints**: Create RESTful APIs for searching articles, retrieving content, updating articles, and handling user authentication.
  + **Version Control**: Store versions of articles in the database to track changes.
  + **Moderation Logic**: Ensure admins can review edits before they go live (optional).
  + **Search Indexing**: Use tools like **Elasticsearch** or **Solr** to optimize search functionality.

**3. Database Design**

**a. Tables/Collections:**

* **Articles Table**:
  + **Fields**:
    - id: Unique identifier.
    - title: Title of the article.
    - content: The main text (can use TEXT type or store as JSON if rich text).
    - author\_id: The ID of the user who created the article.
    - created\_at: Timestamp of article creation.
    - updated\_at: Timestamp of the last update.
* **Article Versions Table**:
  + **Fields**:
    - id: Unique identifier.
    - article\_id: Reference to the article.
    - version\_number: Version count (incremental).
    - editor\_id: The ID of the user who made the edit.
    - content: The actual content of the version.
    - edit\_summary: A short description of the changes made.
    - created\_at: Timestamp of version creation.
* **Users Table**:
  + **Fields**:
    - id: Unique user ID.
    - username: Username of the user.
    - email: Email address.
    - role: Role of the user (e.g., reader, editor, admin).
    - password\_hash: Encrypted password for login.
* **Search Index (Optional)**:
  + **Fields**:
    - id: Index for faster searching (could be powered by **Elasticsearch** or **Solr**).
    - keywords: Searchable terms extracted from the title, content, etc.
    - article\_id: Reference to the article.

**b. Relationships:**

* **Article to Versions**: One-to-Many (each article has many versions).
* **Users to Articles**: Many-to-Many (a user can edit multiple articles, and multiple users can edit a single article).

**4. Version Control System**

**Basic Versioning:**

Every time an article is updated, save a new version in the Article Versions table while keeping the original article in the Articles table. This allows for tracking changes and reverting to previous versions.

sql

Copy code

Articles Table

-----------------------------------

id | title | content | author\_id | updated\_at

Article Versions Table

-----------------------------------

id | article\_id | version\_number | editor\_id | content | created\_at

**Edit Conflict Resolution:**

* Use "locking" mechanisms (optimistic/pessimistic locking) to prevent two users from editing the same article simultaneously.
* If a conflict occurs, notify the users and suggest resolving conflicts manually (like in Wikipedia).

**5. Search Functionality**

* **Basic Search**: Implement basic search using SQL queries (LIKE %keyword%) to match titles and content.
* **Advanced Search**: For large-scale applications, integrate **Elasticsearch** or **Solr** for full-text search capabilities.
  + **Indexing**: Whenever an article is updated or created, re-index the content in the search engine.
  + **Querying**: Allow users to query the search engine for relevant topics.

**6. User Authentication and Authorization**

**Authentication:**

* Implement login and registration using **JWT (JSON Web Token)** or **OAuth**.
* Users can log in, create accounts, and have persistent sessions.

**Authorization:**

* Roles:
  + **Reader**: Can only view articles.
  + **Editor**: Can view and edit articles.
  + **Admin**: Can view, edit, and moderate articles (approve/reject edits, restore versions).
* Based on roles, restrict access to certain API endpoints (e.g., only admins can moderate).

**7. Moderation and Approval System (Optional)**

For sites where content quality is critical, you may want a moderation system:

* **Moderation Queue**: When an article is edited, the changes go to a "pending approval" queue.
* **Admin Review**: Admins can review and approve or reject edits before they go live.
* **Notifications**: Users can be notified when their changes are approved or rejected.

**8. Rich Text Editing**

Use a WYSIWYG (What You See Is What You Get) editor to allow users to edit text with formatting options (bold, italic, headings, links, etc.).

* Popular rich text editors include **Quill**, **CKEditor**, and **TinyMCE**.
* **Markdown Support**: Alternatively, allow users to write articles in **Markdown** and convert it to HTML.

**9. Notifications and Real-time Updates**

* **Real-time Notifications**: Notify users of changes to the articles they are following or edits they made.
* **WebSocket or Polling**: Use WebSockets for real-time notifications (e.g., when a user updates an article) or implement periodic polling.

**10. Security Considerations**

* **Input Validation and Sanitization**: Ensure all user input is sanitized to prevent **XSS (Cross-Site Scripting)** and other vulnerabilities.
* **Rate Limiting**: Protect against brute force attacks on login pages by implementing rate-limiting.
* **Data Encryption**: Store user passwords securely using a strong hash (e.g., **bcrypt**).
* **Authorization Checks**: Ensure proper authorization checks for all endpoints.

**11. Scalability and Performance**

* **Caching**: Use **Redis** or **Memcached** to cache frequently accessed articles.
* **CDN**: Serve static files and assets via a CDN (Content Delivery Network) for faster page load times.
* **Database Optimization**: Index key columns (like title, id, and updated\_at) in the database to speed up queries.