Steps to Develop the Youth-Led News Platform:

1. Define the Core Requirements

- **User Registration & Login**: Users can sign up (with age restrictions) and create an account. Login for returning users.
- **User Roles**: Define three main roles: Regular User, Admin, and Super Admin.
    - **Regular User**: Can post news, share ideas, and view content.
    - **Admin**: Approves or rejects submitted news, manages the content, and ensures proper investigation before publishing.
    - **Super Admin**: Manages admins and overall application security and maintenance.
- **Dashboard**:
    - **User Dashboard**: Allows users to submit articles, view drafts, manage their submissions, and track the status (pending, approved, rejected).
    - **Admin Dashboard**: Allows admins to view submitted news, approve/reject posts, investigate and manage the platform.
    - **Super Admin Dashboard**: Super Admin can monitor the overall health of the platform, user statistics, and content flow.
- **Content Submission**: Users can submit text content (via rich text editor), audio, video, and images.
- **Moderation**: Admin must review all submissions before they are published. Rejected content should include feedback for the user.
- **Responsive Design**: Ensure the website is mobile-friendly and accessible.
- **Analytics**: Track user engagement (e.g., number of posts, views, likes).

2. Organize the Development Process

- **Technology Stack**:
    - **Frontend**: Next.js (React) for scalability and SEO optimization.
    - **Backend**: Node.js with Strapi (Headless CMS) for handling content, or a custom Node.js/Express server.
    - **Database**: MongoDB or PostgreSQL for managing user data and news content.
    - **State Management**: Zustand for managing the application state (user sessions, submissions, etc.).
    - **Rich Text Editor**: A robust editor like Quill.js or TinyMCE to allow rich text formatting.
    - **File Uploads**: Handle media (audio, video, images) with cloud services like AWS S3 or Cloudinary.
    - **Authentication**: Use JWT or OAuth for login functionality.
    - **Admin Panel**: Strapi offers a built-in admin panel, or you can build a custom panel using React Admin.

3. Requirement Document Structure

For each page, define the following:

- **Home Page**
  - **Purpose**: Welcome users, showcase the latest approved articles, and encourage new submissions.
  - **Components**:
    - Latest News Section: Displays featured articles.
    - Trending News Section: Displays popular posts.
    - Call-to-Action (CTA): Invite users to submit their news/ideas.
  - **User Interaction**: Users can browse articles, click to view details, and sign up or log in.
- **User Dashboard**:
  - **Purpose**: Give users control over their submissions and profile.
  - **Components**:
    - Submission Form: Rich text editor for news content, with options for audio, video, and images.
    - Status View: Shows a list of submitted articles with statuses (draft, submitted, approved, rejected).
    - Edit Drafts: Users can edit articles before submission.
    - Submission History: View of all previously submitted news.
  - **User Interaction**: Users submit articles, track status, and edit drafts.
- **Admin Dashboard**:
  - **Purpose**: Provide tools for admins to manage and moderate content.
  - **Components**:
    - Pending Submissions: List of user-submitted articles waiting for approval.
    - Investigation Tools: Functionality to research the validity of submitted content (could link to external sources).
    - Approve/Reject: Admin can approve or reject a submission, with the option to provide feedback.
    - Dashboard Analytics: Overview of the platform's performance, including number of submissions, approvals, and engagement.
  - **Admin Interaction**: Admins investigate submissions, approve or reject, and provide feedback.
- **Super Admin Dashboard**:
  - **Purpose**: Oversee overall platform functionality and manage users/admins.
  - **Components**:
    - Admin Management: Add, remove, or manage admins.
    - Platform Monitoring: Analytics and insights into platform performance, content flow, user activity.
  - **Super Admin Interaction**: Monitor platform health, manage admin privileges.

4. Design Considerations

- **Target Audience Design**: Since the target audience is students and young people (15-75 years old), the design should be vibrant, engaging, and easy to navigate. Use colors, fonts, and layouts that appeal to younger generations.
- **User Experience (UX)**:
    - Keep the submission process intuitive and simple.
    - The design should be responsive and optimized for both desktop and mobile users.
- **Rich Media Integration**: Ensure seamless upload and playback of audio, video, and images.
- **Social Sharing**: Integrate social media sharing options to allow users to promote their articles.

5. Create a Roadmap

- **Phase 1**: User registration, submission form, and content moderation system.
- **Phase 2**: User dashboards, rich media support (audio, video, images), and admin panel.
- **Phase 3**: Super admin features, advanced analytics, and social media integration.
- **Phase 4**: Launch with beta testers to gather feedback and refine the platform

Here are the key collections and fields, along with relationships:

1. User Collection (Strapi Built-in)

This collection will store user information. Strapi provides this out of the box with customizable fields.

Fields:

- `username` (String, unique): The username of the user.
- `email` (Email, unique): User email for authentication.
- `password` (String): User password (hashed).
- `role` (Enum): The role of the user (Regular User, Admin, Super Admin).
- `created_at` (Date): Timestamp of when the user was created.

Relationships:

- `articles`: One-to-Many relation with the **Article** collection (a user can create many articles).
- `comments`: One-to-Many relation with the **Comment** collection (a user can comment on many articles).

2. Article Collection

This collection stores articles (posts) shared by users. The articles can contain text, media, and be in various states (draft, submitted, approved, rejected).

Fields:

- `title` (String, required): The title of the article.
- `content` (RichText, required): The body of the article (supports rich text with formatting, links, etc.).
- `media` (Media - files): Allows users to upload images, videos, or audio.
- `status` (Enum, default: "draft"): The status of the article (`draft`, `submitted`, `approved`, `rejected`).
- `rejection_reason` (String): If rejected, a reason can be provided by the admin.
- `submitted_at` (Date): Timestamp for when the user submits the article for review.
- `published_at` (Date): Timestamp for when the article is approved and published.
- `views` (Number, default: 0): A count of how many times the article has been viewed.
- `likes` (Number, default: 0): A count of likes from users.
- `comments` (Relation): One-to-Many relation to the **Comment** collection.

Relationships:

- `author`: Many-to-One relation with the **User** collection (each article belongs to one author).
- `reviewed_by`: Many-to-One relation with the **User** collection (each article is reviewed by one admin).

3. Comment Collection

This collection stores user comments on articles.

Fields:

- `content` (String, required): The text content of the comment.
- `created_at` (Date): Timestamp of when the comment was created.

Relationships:

- `article`: Many-to-One relation with the **Article** collection (a comment belongs to one article).
- `user`: Many-to-One relation with the **User** collection (a comment belongs to one user).

4. Admin Review Collection

This collection tracks the review process of articles by admins (to add an audit trail).

Fields:

- `review_notes` (String, required): Notes by the admin about the review decision (why approved or rejected).
- `action_taken` (Enum, required): The action taken by the admin (`approved`, `rejected`).
- `review_date` (Date): Timestamp of when the admin reviewed the article.

Relationships:

- `admin`: Many-to-One relation with the **User** collection (each review is performed by one admin).
- `article`: Many-to-One relation with the **Article** collection (each review belongs to one article).

5. Media Collection

This collection will handle the media (audio, video, images) that users upload in their articles.

Fields:

- `file` (Media, required): The uploaded media file (image, video, audio).
- `type` (Enum, required): The type of media (`image`, `video`, `audio`).
- `uploaded_at` (Date): Timestamp of when the media was uploaded.

Relationships:

- `article`: Many-to-One relation with the **Article** collection (each media belongs to one article).

6. Role Collection (Strapi Built-in)

Strapi comes with built-in roles (like Admin, Super Admin, and Regular User). You can extend this for different user permissions.

Fields:

- `name` (String, required): The name of the role (e.g., Admin, Super Admin, Regular User).
- `description` (String, optional): A description of the role.

7. Notification Collection (Optional)

This collection can be used to notify users (or admins) about events, such as article approval, rejection, or feedback.

Fields:

- `message` (String, required): The content of the notification.
- `type` (Enum, required): The type of notification (`article_approved`, `article_rejected`, etc.).
- `is_read` (Boolean, default: false): Whether the notification has been read.
- `created_at` (Date): Timestamp of when the notification was created.

Relationships:

- `user`: Many-to-One relation with the **User** collection (each notification belongs to one user).
- `article`: Many-to-One relation with the **Article** collection (notifications can be tied to specific articles).

Overall Relationships:

1. **User** (Regular User, Admin, Super Admin)
    - Can post **Articles**.
    - Can comment on **Articles**.
    - Can review **Articles** (Admin role).
    - Can manage users (Super Admin role).
2. **Article**
    - Belongs to a **User** (Author).
    - Can have many **Comments**.
    - Can be reviewed by a **User** (Admin).
    - Can have many **Media** items (Images, Videos, Audio).
3. **Comment**
    - Belongs to a **User** (Commenter).
    - Belongs to an **Article**.
4. **Media**
    - Belongs to an **Article**.
5. **Admin Review**
    - Belongs to a **User** (Admin).
    - Belongs to an **Article**.

Database Example:

This schema results in a relational structure where each collection relates to the others. For example:

- A user can write many articles, but each article has one author.
- An article can have multiple comments and media files, while being reviewed by a single admin.
- Each review is associated with an article and an admin.

How This Works in Strapi:

- **User Roles & Permissions**: Strapi has a built-in role management system, so you can define who has access to what (Admins review content, Regular users submit content).
- **Content Types**: Strapi allows you to define content types, and all collections mentioned above can be configured as content types (User, Article, Comment, Media, Review, etc.).
- **Media Management**: Strapi handles media uploads and integrates with cloud providers (AWS, Cloudinary, etc.) for hosting images, videos, and audio.
- **Rich Text**: The rich text editor will be integrated within the article submission form, allowing users to format their content before submitting.

This schema is scalable and can be extended as the platform grows (e.g., adding more moderation features, or notifications).