Next.js Application Documentation

Table of Contents

Overview

This is a Next.js 16 application with MongoDB database, featuring:

- Dual authentication (Credentials + Google OAuth)
- Post management system with voting functionality
- Protected routes for authenticated users
- RESTful API architecture

Tech Stack:

- **Framework:** Next.js 16.0.3 with App Router
- **Database:** MongoDB with Mongoose ODM
- **Authentication:** NextAuth.js v4.24.13
- **Password Hashing:** bcrypt/bcryptjs
- **UI:** React 19, Tailwind CSS, Radix UI components

Environment Setup

Required Environment Variables

Create a `.env.local` file in your project root:

```env
# MongoDB Connection
MONGODB_URI=mongodb+srv://username:password@cluster.mongodb.net/database_name

# NextAuth Configuration
NEXTAUTH_SECRET=your-secret-key-here
NEXTAUTH_URL=http://localhost:3000

# Google OAuth (Optional)
GOOGLE_CLIENT_ID=your-google-client-id
GOOGLE_CLIENT_SECRET=your-google-client-secret
```

Installation

```bash
npm install
npm run dev
```

Database Architecture

Connection Management (`src/lib/mongo.js`)

The application uses a connection caching strategy to prevent multiple connections in development:

```javascript
// Singleton pattern for MongoDB connection
let cached = global.mongoose;

if (!cached) {
  cached = global.mongoose = { conn: null, promise: null };
}
```

Features:

- Connection reuse across requests
- Automatic error handling
- Console logging for connection status

Database Models

User Model (`src/models/User.js`)

```javascript
{
  name: String,          // Required, trimmed
  email: String,         // Required, unique, lowercase
  password: String,      // Optional (null for Google OAuth users)
  provider: String,      // "credentials" | "google" (default: "credentials")
  googleId: String,      // Unique Google ID (sparse index)
  createdAt: Date,       // Auto-generated
  updatedAt: Date        // Auto-generated
}
```

Indexes:

- `email`: Unique index
- `googleId`: Unique sparse index (allows null values)

Post Model (`src/models/Post.js`)

```javascript
{
  title: String,              // Required, trimmed
  description: String,        // Required, trimmed
  author: ObjectId,           // Reference to User, required
  authorEmail: String,        // Required
  upvoters: [ObjectId],       // Array of User IDs who upvoted
  downvoters: [ObjectId],     // Array of User IDs who downvoted
  voteCount: Number,          // Default: 0
  createdAt: Date,            // Auto-generated
  updatedAt: Date             // Auto-generated
}
```

Relationships:

- `author` → References `User._id`
- `upvoters` → Array of references to `User._id`
- `downvoters` → Array of references to `User._id`

Authentication System

NextAuth Configuration (`src/lib/auth.js`)

The application supports two authentication methods:

1. Credentials Authentication (Email/Password)

Flow:

1. User submits email and password
2. System queries MongoDB for user by email
3. Validates password using bcrypt comparison
4. Returns user object if successful

Password Validation:

- Checks if user exists
- Verifies password field exists (prevents Google users from logging in with credentials)
- Compares hashed password with bcrypt

---

2. Google OAuth Authentication

Configuration:

```javascript
GoogleProvider({
  clientId: process.env.GOOGLE_CLIENT_ID,
  clientSecret: process.env.GOOGLE_CLIENT_SECRET,
  authorization: {
    params: {
      prompt: "consent",
      access_type: "offline",
      response_type: "code"
    }
  }
})
```

Flow:

1. User clicks "Sign in with Google"
2. Redirected to Google OAuth consent screen
3. On successful authentication, `signIn` callback is triggered
4. System checks if user exists in database
5. If new user, creates account with `provider: "google"`
6. User ID is attached to session

Authentication Callbacks

`signIn` Callback

- **Purpose:** Handle Google OAuth user creation/login
- **Process:**
  - Checks if user exists by email
  - Creates new user if doesn't exist
  - Attaches MongoDB `_id` to user object

`jwt` Callback

- **Purpose:** Add custom data to JWT token
- **Data Added:**
  - `token.id`: MongoDB user ID
  - `token.provider`: Authentication provider

`session` Callback

- **Purpose:** Make JWT data available in session object
- **Data Exposed:**
  - `session.user.id`: User's MongoDB ID
  - `session.user.provider`: Authentication method used

Session Strategy

```javascript
session: { strategy: "jwt" }
```

- Uses JWT tokens (stateless)
- No database queries for session validation
- Token stored in HTTP-only cookie

API Endpoints

Base URL

```
http://localhost:3000/api
```

Authentication Endpoints

POST /api/register

Register a new user with credentials.

Request Body:

```json
{
  "name": "John Doe",
  "email": "john@example.com",
  "password": "securePassword123"
}
```

Success Response (200):

```json
{
  "ok": true,
  "userId": "507f1f77bcf86cd799439011"
}
```

Error Responses:

Status	Response	Reason
400	`{ "error": "Missing fields" }`	Email or password not provided
409	`{ "error": "Email already used" }`	Email already registered

Process:

1. Validates required fields
2. Checks for existing email
3. Hashes password with bcrypt (10 rounds)
4. Creates user with default name if not provided
5. Returns user ID

POST /api/login

Login with credentials (manual endpoint, not used by NextAuth).

Request Body:

```
json

{
  "email": "john@example.com",
  "password": "securePassword123"
}
```

Success Response (200):

```
json

{
  "success": true,
  "message": "Login successful",
  "user": {
    "id": "507f1f77bcf86cd799439011",
    "name": "John Doe",
    "email": "john@example.com"
  }
}
```

Error Responses:

Status	Response	Reason
400	`{ "error": "User not found" }`	Email doesn't exist
400	`{ "error": "Wrong password" }`	Password incorrect
500	`{ "error": "Server error" }`	Database or server error

GET/POST /api/auth/[...nextauth]

NextAuth.js authentication handler (managed automatically).

Endpoints Provided:

- `GET /api/auth/signin` - Sign in page
- `POST /api/auth/signin/credentials` - Credentials login
- `GET /api/auth/callback/google` - Google OAuth callback
- `GET /api/auth/signout` - Sign out
- `GET /api/auth/session` - Get current session

Post Management Endpoints

POST /api/posts

Create a new post (requires authentication).

Headers:

```
Cookie: next-auth.session-token=...
```

Request Body:

```
json

{
  "title": "My First Post",
  "description": "This is the content of my post"
}
```

Success Response (200):

```json
{
  "success": true,
  "post": {
    "_id": "507f1f77bcf86cd799439011",
    "title": "My First Post",
    "description": "This is the content of my post",
    "author": "507f1f77bcf86cd799439012",
    "authorEmail": "john@example.com",
    "upvoters": [],
    "downvoters": [],
    "voteCount": 0,
    "createdAt": "2025-01-15T10:30:00.000Z",
    "updatedAt": "2025-01-15T10:30:00.000Z"
  }
}
```

Error Responses:

Status	Response	Reason
401	`{ "success": false, "error": "Unauthorized. Please log in." }`	No valid session
400	`{ "success": false, "error": "Title and description are required" }`	Missing fields
500	`{ "success": false, "error": "..." }`	Server error

Process:

1. Validates session using `getServerSession()`
2. Extracts user ID from session
3. Validates required fields
4. Trims whitespace from title and description
5. Creates post with author information

GET /api/posts

Fetch posts with pagination and filtering.

Query Parameters:

Parameter	Type	Default	Description
`page`	number	1	Page number
`limit`	number	6	Posts per page
`userOnly`	boolean	false	Filter by current user's posts

Example Requests:

```
GET /api/posts
GET /api/posts?page=2&limit=10
GET /api/posts?userOnly=true
```

Success Response (200):

```json
{
  "success": true,
  "posts": [
    {
      "_id": "507f1f77bcf86cd799439011",
      "title": "Post Title",
      "description": "Post content",
      "author": "507f1f77bcf86cd799439012",
      "authorEmail": "john@example.com",
      "upvoters": ["507f1f77bcf86cd799439013"],
      "downvoters": [],
      "voteCount": 1,
      "createdAt": "2025-01-15T10:30:00.000Z",
      "updatedAt": "2025-01-15T10:30:00.000Z"
    }
  ],
  "totalPosts": 25,
  "totalPages": 5,
  "currentPage": 1
}
```

Error Response (401) - when `userOnly=true` and not authenticated:

```json
```

```
{
  "success": false,
  "error": "Unauthorized"
}
```

Features:

- Sorted by creation date (newest first)
- Pagination support
- User-specific filtering requires authentication
- Returns lean documents (plain objects, not Mongoose documents)

GET /api/posts/[id]

Fetch a single post by ID.

Example Request:

```
GET /api/posts/507f1f77bcf86cd799439011
```

Success Response (200):

```
json

{
  "success": true,
  "post": {
    "_id": "507f1f77bcf86cd799439011",
    "title": "Post Title",
    "description": "Post content",
    "author": "507f1f77bcf86cd799439012",
    "authorEmail": "john@example.com",
    "upvoters": [],
    "downvoters": [],
    "voteCount": 0,
    "createdAt": "2025-01-15T10:30:00.000Z",
    "updatedAt": "2025-01-15T10:30:00.000Z"
  }
}
```

Error Responses:

Status	Response	Reason
404	`{ "success": false, "error": "Post not found" }`	Invalid ID or post doesn't exist
500	`{ "success": false, "error": "..." }`	Server error

PUT /api/posts/[id]

Update an existing post.

Example Request:

```
PUT /api/posts/507f1f77bcf86cd799439011
```

Request Body:

```
json

{
  "title": "Updated Title",
  "description": "Updated content"
}
```

Success Response (200):

```
json

{
  "success": true,
  "post": {
    "_id": "507f1f77bcf86cd799439011",
    "title": "Updated Title",
    "description": "Updated content",
    "author": "507f1f77bcf86cd799439012",
    "authorEmail": "john@example.com",
    "upvoters": [],
    "downvoters": [],
    "voteCount": 0,
    "createdAt": "2025-01-15T10:30:00.000Z",
    "updatedAt": "2025-01-15T11:45:00.000Z"
  }
}
```

Error Responses:

Status	Response	Reason
404	`{ "success": false, "error": "Post not found" }`	Invalid ID or post doesn't exist
500	`{ "success": false, "error": "..." }`	Server or validation error

Process:

1. Finds post by ID
2. Updates title and description
3. Runs Mongoose validators
4. Returns updated document
5. Updates `updatedAt` timestamp automatically

DELETE /api/posts/[id]

Delete a post.

Example Request:

```
DELETE /api/posts/507f1f77bcf86cd799439011
```

Success Response (200):

```json
{
  "success": true
}
```

Error Responses:

Status	Response	Reason
404	`{ "success": false, "error": "Post not found" }`	Invalid ID or post doesn't exist
500	`{ "success": false, "error": "..." }`	Server error

POST /api/posts/vote

Vote on a post (upvote/downvote).

Request Body:

```json
{
  "postId": "507f1f77bcf86cd799439011",
  "userId": "507f1f77bcf86cd799439012",
  "voteType": "up"
}
```

Vote Types:

- `"up"` - Upvote the post
- `"down"` - Downvote the post
- `""` (empty string) - Remove vote

Success Response (200):

```json
{
  "success": true,
  "post": {
    "_id": "507f1f77bcf86cd799439011",
    "title": "Post Title",
    "description": "Post content",
    "author": "507f1f77bcf86cd799439013",
    "authorEmail": "author@example.com",
    "upvoters": ["507f1f77bcf86cd799439012"],
    "downvoters": [],
    "voteCount": 1,
    "createdAt": "2025-01-15T10:30:00.000Z",
    "updatedAt": "2025-01-15T12:00:00.000Z"
  }
}
```

Error Response (404):

```json
```

```
{
  "error": "Post not found"
}
```

Voting Logic

The voting system prevents duplicate votes and handles vote changes:

Upvote Logic (`voteType: "up"`)

1. **User already upvoted:**
   - Remove user from `upvoters`
   - Decrease `voteCount` by 1
   - (Acts as toggle - removes upvote)
2. **User already downvoted:**
   - Remove user from `downvoters`
   - Increase `voteCount` by 1
   - Add user to `upvoters`
   - Increase `voteCount` by 1
   - (Net effect: +2 to voteCount)
3. **User hasn't voted:**
   - Add user to `upvoters`
   - Increase `voteCount` by 1

Downvote Logic (`voteType: "down"`)

1. **User already downvoted:**
   - Remove user from `downvoters`
   - Increase `voteCount` by 1
   - (Acts as toggle - removes downvote)
2. **User already upvoted:**
   - Remove user from `upvoters`
   - Decrease `voteCount` by 1
   - Add user to `downvoters`
   - Decrease `voteCount` by 1
   - (Net effect: -2 to voteCount)
3. **User hasn't voted:**
   - Add user to `downvoters`
   - Decrease `voteCount` by 1

Remove Vote Logic (`voteType: ""`)

1. **User has upvoted:**
   - Remove user from `upvoters`
   - Decrease `voteCount` by 1
2. **User has downvoted:**
   - Remove user from `downvoters`
   - Increase `voteCount` by 1

Middleware & Route Protection

Protected Routes (`proxy.js`)

The middleware protects specific routes from unauthorized access.

Protected Routes:

- `/dashboard/*`
- `/add-post/*`
- `/edit-post/*`
- `/manage-posts/*`

How It Works:

1. Middleware intercepts requests to protected routes
2. Checks for valid JWT token using `getToken()`
3. If no token found, redirects to `/login`
4. If token valid, allows request to proceed

Code:

```javascript
export const config = {
  matcher: [
    "/dashboard/:path*",
    "/add-post/:path*",
    "/edit-post/:path*",
    "/manage-posts/:path*"
  ],
};

export default async function proxy(req) {
  const token = await getToken({
    req,
    secret: process.env.NEXTAUTH_SECRET,
  });

  if (!token) {
    return NextResponse.redirect(new URL("/login", req.url));
  }

  return NextResponse.next();
}
```

Security Features

1. Password Security

Hashing Algorithm: bcrypt

- **Rounds:** 10 (salt rounds)
- **Process:** Passwords are hashed before storage
- **Validation:** Uses `bcrypt.compare()` for verification

```javascript
// Registration
const hashed = await bcrypt.hash(password, 10);

// Login
const isValid = await bcrypt.compare(credentials.password, user.password);
```

---

2. Session Security

JWT Configuration:

- Stored in HTTP-only cookies (not accessible via JavaScript)
- Signed with `NEXTAUTH_SECRET`
- Token includes minimal data (user ID, provider)

Session Validation:

- All protected API endpoints use `getServerSession()`
- Middleware validates tokens on protected routes
- No database queries needed for session validation

---

3. Authentication Flow Security

Credentials Provider:

- Prevents timing attacks with proper error messages
- Checks provider type (prevents Google users from credential login)
- Uses bcrypt for constant-time password comparison

Google OAuth:

- PKCE flow enabled (`response_type: "code"`)
- Offline access for refresh tokens
- Consent prompt ensures user awareness

---

4. Database Security

Connection Security:

- MongoDB connection string stored in environment variables
- Connection caching prevents multiple connections
- Proper error handling without exposing sensitive data

Schema Validation:

- Required fields enforced at database level
- Email uniqueness constraint
- Trimming prevents whitespace attacks
- Lowercase email normalization

---

5. API Security

Input Validation:

- Required field checks on all POST endpoints
- Whitespace trimming on user inputs
- Mongoose schema validators

Error Handling:

- Generic error messages (no sensitive data leakage)
- Proper HTTP status codes
- Console logging for debugging (server-side only)

Authorization:

- Session validation on protected endpoints
- User ID from session (not from request body)
- Ownership validation for user-specific operations

Common Usage Patterns

Client-Side Authentication Check

```javascript
import { useSession } from "next-auth/react";

export default function Component() {
  const { data: session, status } = useSession();

  if (status === "loading") {
    return <div>Loading...</div>;
  }

  if (status === "unauthenticated") {
    return <div>Please sign in</div>;
  }

  return <div>Welcome {session.user.name}</div>;
}
```

Server-Side Authentication Check

```javascript
import { getServerSession } from "next-auth";
import { authOptions } from "@/lib/auth";

export default async function Page() {
  const session = await getServerSession(authOptions);

  if (!session) {
    redirect("/login");
  }

  return <div>Protected Content</div>;
}
```

Making Authenticated API Calls

```javascript
```

```
// Client component
async function createPost(title, description) {
  const response = await fetch("/api/posts", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({ title, description }),
  });

  const data = await response.json();
  return data;
}
```

Note: Session cookie is automatically included with fetch requests.

Fetching User's Posts

```
javascript

async function fetchMyPosts() {
  const response = await fetch("/api/posts?userOnly=true");
  const data = await response.json();
  return data.posts;
}
```

Voting on Posts

```
javascript

async function voteOnPost(postId, userId, voteType) {
  const response = await fetch("/api/posts/vote", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({ postId, userId, voteType }),
  });

  const data = await response.json();
  return data.post;
}

// Usage
await voteOnPost("507f...", "507f...", "up");    // Upvote
await voteOnPost("507f...", "507f...", "down");  // Downvote
await voteOnPost("507f...", "507f...", "");      // Remove vote
```

Troubleshooting

Common Issues

1. "MongoDB Connection Error"

Cause: Invalid `MONGODB_URI` or network issue

Solution:

- Verify MongoDB connection string
- Check MongoDB Atlas IP whitelist
- Ensure database user has proper permissions

---

2. "No user found" during login

Cause: User trying to log in with credentials but registered with Google

Solution:

- The system checks for password field
- Displays "Please sign in with Google" message
- User should use Google OAuth button

---

3. "Unauthorized" on API calls

Cause: No valid session or expired token

Solution:

- Check if user is logged in
- Verify `NEXTAUTH_SECRET` matches between deployment and local
- Clear cookies and re-login

---

4. Voting not working

Cause: User ID not passed correctly

Solution:

- Ensure `session.user.id` is being sent
- Verify session callback is adding ID to session
- Check MongoDB IDs are valid ObjectIds

Development Notes

Unused Files

The following hooks exist but are not currently implemented:

- `src/hooks/useAuth.js` (empty)
- `src/hooks/usePost.js` (empty)

These could be used to create reusable React hooks for:

- Authentication state management
- Post CRUD operations
- Voting functionality

Potential Improvements

1. **Authorization on Edit/Delete:**
   - Currently no check if user owns the post
   - Should add: `if (post.author.toString() !== session.user.id)`
2. **Rate Limiting:**
   - No rate limiting on API endpoints
   - Consider adding for voting and post creation
3. **Input Sanitization:**
   - Add XSS protection on post content
   - Consider using DOMPurify or similar
4. **Error Logging:**
   - Implement proper error tracking (Sentry, etc.)
   - Replace console.error with logging service
5. **API Response Consistency:**
   - Some endpoints use `{ success: true, ... }`
   - Others use `{ ok: true, ... }`
   - Standardize response format
6. **Password Requirements:**
   - No password strength validation
   - Add minimum length and complexity rules

API Testing with cURL

Register User

```bash
curl -X POST http://localhost:3000/api/register \
  -H "Content-Type: application/json" \
  -d '{"name":"John Doe","email":"john@example.com","password":"password123"}'
```

Create Post

```bash
curl -X POST http://localhost:3000/api/posts \
  -H "Content-Type: application/json" \
  -H "Cookie: next-auth.session-token=YOUR_TOKEN" \
  -d '{"title":"Test Post","description":"This is a test"}'
```

Get All Posts

```bash
curl http://localhost:3000/api/posts?page=1&limit=10
```

Vote on Post

```bash
curl -X POST http://localhost:3000/api/posts/vote \
  -H "Content-Type: application/json" \
  -d '{"postId":"507f1f77bcf86cd799439011","userId":"507f1f77bcf86cd799439012","voteType":"up"}'
```

Conclusion

This application provides a solid foundation for a post management system with dual authentication. The architecture is clean, scalable, and follows Next.js best practices. With the suggested improvements, it can be production-ready.

For questions or issues, refer to:

- Next.js Documentation
- NextAuth.js Documentation
- Mongoose Documentation