Project Overview

Version: 0.1.0
Generated: 7/22/2025

Introduction

This document provides an overview of Vydeo, a feature-rich React-based social video platform. The application allows users to share and consume video content, chat with friends, and engage with a community of content creators. Built on React 18 with Material UI components, the application delivers a responsive and modern user experience on all devices.

Project Architecture

The application follows a modular component-based architecture built with React 18 and uses Redux Toolkit for state management. It implements a single-page application (SPA) model with client-side routing via React Router v6. The application communicates with multiple backend services (Java and Python) via RESTful APIs and uses WebSockets for real-time features such as chat and notifications.

Architecture Diagram:

Key Features

- Secure JWT-based authentication with token refresh mechanism
- Real-time chat with message history, WebSocket notifications, and online status indicators
- Video content management with support for uploads, playback, and comments
- Social features including friends list, activity feeds, and user profiles
- Advanced search capabilities with instant results and filters
- Theme switching between light and dark modes with persistent user preferences
- QR code scanning functionality for quick connections
- Role-based access control for administrative features
- Offline data persistence using IndexedDB

The application integrates with multiple backend services:

- Java API (https://apis.vydeo.xyz/java): Primary backend for core functionality
- Python API (https://apis.vydeo.xyz/py): Specialized services for data processing and AI features
- WebSocket Server (wss://apis.vydeo.xyz/ws): Real-time communication for chat and notifications
- Download Service: Dedicated service for handling large file downloads

Target Audience

The application is designed for content creators, social media users, and video enthusiasts:

- Content creators who want to share videos and build a following
- Users seeking to discover and engage with video content
- Social media users looking for real-time interaction with friends
- Communities centered around specific video content categories

Application Modules

The application is organized into several key functional modules:

- Authentication: User registration, login, and account management
- Video Platform: Video browsing, playback, and interaction
- Social Network: Friend management, activities, and profile features
- Messaging: Real-time chat with individuals and groups
- Content Management: Upload, edit, and manage video content
- Search: Advanced search capabilities across all content types
- Settings: User preferences and application configuration
- Admin Tools: User management and system administration

API Documentation

Version: 0.1.0
Generated: 7/22/2025

API Overview

This document provides detailed specifications for the RESTful API endpoints used by the Blog Web Application. The API is hosted at https://api.blogapp.com/v1 and uses JWT authentication with rate limiting of 100 requests per minute per user.

API Standards

All API responses follow a standard format with appropriate HTTP status codes:

```
{
  "code": 200, // HTTP status code
  "success": true, // boolean indicating success
  "message": "Operation successful", // human-readable message
  "data": {}, // response payload
  "timestamp": "2023-07-15T12:34:56Z" // ISO timestamp
}
```

Authentication Endpoints

Login

Endpoint: /auth/login
Method: POST
Description: Authenticates a user and returns a JWT token with 24-hour validity
Rate Limit: 10 attempts per minute per IP
Request Body:

```
{
  "username": "string", // Email or username
  "password": "string", // Min 8 characters
  "rememberMe": boolean, // Optional, default: false
  "deviceInfo": { // Device information for security
    "deviceId": "string",
    "platform": "string",
    "browserName": "string"
  }
}
```

Response:

```
{
  "code": 200,
  "success": true,
  "message": "Login successful",
  "data": {
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "expiresIn": 86400,
    "userId": "u123456789",
    "userRole": "user",
    "firstName": "John",
    "lastName": "Doe",
    "profileImage": "https://cdn.blogapp.com/profiles/john-doe.jpg"
  }
}
```

Sign Up

Endpoint: /auth/signup
Method: POST

Description: Registers a new user with email verification
Rate Limit: 5 attempts per hour per IP
Request Body:
```
{
  "username": "string", // 3-20 alphanumeric characters
  "email": "string", // Valid email format
  "password": "string", // Min 8 chars, 1 uppercase, 1 number, 1 special
  "firstName": "string",
  "lastName": "string",
  "termsAccepted": boolean, // Must be true
  "marketingConsent": boolean // Optional
}
```
Response:
```
{
  "code": 201,
  "success": true,
  "message": "User registered successfully. Please verify your email.",
  "data": {
    "userId": "u123456789",
    "verificationSent": true,
    "verificationExpiry": "2023-07-15T14:34:56Z"
  }
}
```

Refresh Token
Endpoint: /auth/refresh
Method: POST
Description: Obtains a new access token using a refresh token
Request Body:
```
{
  "refreshToken": "string" // Valid refresh token
}
```
Response:
```
{
  "code": 200,
  "success": true,
  "message": "Token refreshed successfully",
  "data": {
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "expiresIn": 86400
  }
}
```

Blog Post Endpoints

Get All Posts
Endpoint: /posts
Method: GET
Description: Retrieves paginated blog posts with optional filtering
Authentication: Optional - Authenticated users receive personalized results
Query Parameters:
- page: number (default: 1) - Current page number
- limit: number (default: 10, max: 50) - Items per page
- sort: string (default: "createdAt") - Sort field (createdAt, title, likes)
- order: string (default: "desc") - Sort order (asc, desc)
- category: string - Filter by category
- tag: string - Filter by tag
- author: string - Filter by author ID
- search: string - Search in title and content

Response:
```
{
  "code": 200,
  "success": true,
  "message": "Posts retrieved successfully",
```

```
  "data": {
    "posts": [
      {
        "id": "p123456789",
        "title": "Getting Started with React 18",
        "excerpt": "Learn the basics of React 18 and its new features",
        "author": {
          "id": "u987654321",
          "username": "reactmaster",
          "profileImage": "https://cdn.blogapp.com/profiles/reactmaster.jpg"
        },
        "coverImage": "https://cdn.blogapp.com/posts/react18-cover.jpg",
        "createdAt": "2023-07-10T08:15:30Z",
        "updatedAt": "2023-07-10T10:22:15Z",
        "category": "Programming",
        "tags": ["React", "JavaScript", "Frontend"],
        "readTime": 8,
        "likes": 124,
        "comments": 18,
        "isLiked": true, // Only present for authenticated users
        "isBookmarked": false // Only present for authenticated users
      },
      // More posts...
    ],
    "pagination": {
      "total": 142,
      "pages": 15,
      "currentPage": 1,
      "hasNext": true,
      "hasPrev": false
    }
  }
}
```

Create Blog Post

Endpoint: /posts
Method: POST
Description: Creates a new blog post
Authentication: Required (User or Admin role)
Request Body:

```
{
  "title": "string", // 10-150 characters
  "content": "string", // HTML content, min 100 characters
  "excerpt": "string", // Optional, max 300 characters
  "coverImage": "string", // Optional, URL or base64
  "category": "string", // Must be one of predefined categories
  "tags": ["string"], // Optional, max 5 tags
  "status": "string", // "draft" or "published"
  "seoTitle": "string", // Optional, max 70 characters
  "seoDescription": "string", // Optional, max 160 characters
  "allowComments": boolean // Optional, default: true
}
```

Response:

```
{
  "code": 201,
  "success": true,
  "message": "Blog post created successfully",
  "data": {
    "id": "p123456789",
    "title": "Getting Started with React 18",
    "slug": "getting-started-with-react-18",
    "url": "https://blogapp.com/posts/getting-started-with-react-18",
    "status": "published",
    "createdAt": "2023-07-15T12:34:56Z"
  }
}
```

Comment Endpoints

Get Comments for Post

Endpoint: /posts/{postId}/comments
Method: GET
Description: Retrieves comments for a specific blog post with pagination
Path Parameters: postId - ID of the blog post
Query Parameters: page, limit, sort (createdAt, likes), order (asc, desc)
Response Structure: Paginated list of comments with nested replies

User Guide

Version: 0.1.0
Generated: 7/22/2025

Introduction

This user guide provides comprehensive instructions on using the Blog Web Application, a feature-rich platform designed for content creators and readers. Version 2.3.0 introduces several new features and improvements over the previous version.

System Requirements

The application is compatible with:

- Desktop: Chrome 90+, Firefox 88+, Safari 14+, Edge 90+
- Mobile: iOS Safari 14+, Android Chrome 90+
- Minimum screen resolution: 320px width (responsive design)
- JavaScript must be enabled
- Cookies must be enabled for authentication

Getting Started

To begin using the Blog Web Application, you need to create an account or log in with an existing one. The application supports both standard email/password authentication and social login options.

Creating an Account

To create a new account:

- Navigate to https://blogapp.com/signup or tap the "Sign Up" button on the login screen.
- Enter your username (3-20 characters, alphanumeric only).
- Enter your email address (will require verification).
- Create a strong password (minimum 8 characters with at least one uppercase letter, one number, and one special character).
- Review and accept the Terms of Service and Privacy Policy.
- Optional: Enable two-factor authentication for enhanced security.
- Click the "Create Account" button.
- Check your email for a verification link and click it to activate your account.

Refer to Figure 1 in the Appendix for a screenshot of the sign-up form.

Logging In

To log in to your account:

- Navigate to https://blogapp.com/login or click "Log In" on the homepage.
- Enter your username or email address.
- Enter your password.
- Optional: Check "Remember me" to stay logged in for 30 days.
- Click the "Log In" button.
- If you have two-factor authentication enabled, enter the verification code from your authenticator app.

Alternative login methods:
- Click "Continue with Google" to log in with your Google account.
- Click "Continue with Apple" to log in with your Apple ID.
- Click "Continue with Twitter" to log in with your Twitter account.

Dashboard

After logging in, you'll be directed to your personalized dashboard, which displays:

- Recommended blog posts based on your reading history and preferences
- Latest posts from authors you follow
- Your reading list (saved articles)
- Your draft posts (if you are a content creator)
- Notifications (comments, likes, new followers)
- Quick access to popular categories

To customize your dashboard:
- Click the "Customize" button in the top-right corner of the dashboard.
- Select which widgets to show/hide and drag to rearrange them.
- Choose your preferred layout (grid, list, or compact).
- Click "Save Changes" to apply your customizations.

Reading Content

Browsing Articles

To discover and read blog posts:
- Use the search bar at the top to find specific content (supports advanced filters).
- Browse categories by clicking on the category tabs in the navigation menu.
- Explore trending topics in the "Trending Now" section.
- View personalized recommendations in the "For You" section.
- Check the "New" section for the latest publications across all categories.

Reading Experience

While reading an article, you can:
- Adjust text size using the "A-" and "A+" buttons in the reading toolbar.
- Switch between light and dark mode using the theme toggle in the top-right corner.
- Save articles to your reading list by clicking the bookmark icon.
- Share articles via email, social media, or copy the link using the share button.
- Highlight text by selecting it and clicking the highlight icon in the popup menu.
- Add private notes to highlighted sections for your reference.
- Enable "Focus Mode" by clicking the distraction-free icon to hide sidebars and comments.

Creating Content

Writing a Blog Post

To create and publish a new blog post:
- Click the "Write" button in the navigation bar.
- Enter a compelling title (10-150 characters).
- Use the rich text editor to write and format your content:
- • Format text using the toolbar (bold, italic, headings, etc.).
- • Insert images by clicking the image icon or drag-and-drop.
- • Add links by selecting text and clicking the link icon.

- • Insert code snippets using the code block option.
- • Add tables, quotes, and horizontal rules as needed.
- Add relevant tags (up to 5) to help readers discover your content.
- Select a category that best fits your post.
- Upload a cover image (recommended size: 1200x630 pixels).
- Write a brief excerpt (max 300 characters) to appear in previews.
- Preview your post by clicking the "Preview" button.
- Click "Save as Draft" to save without publishing.
- Click "Publish" when you're ready to make it live.

Managing Your Account

Profile Settings

To update your profile information:

- Click your profile picture in the top-right corner.
- Select "Settings" from the dropdown menu.
- In the "Profile" tab, you can:
- • Update your profile picture and cover image
- • Edit your display name and username
- • Update your bio and social media links
- • Set your location and preferred language
- Click "Save Changes" to apply your updates.

Notification Settings

To customize your notification preferences:

- Go to Settings > Notifications.
- Choose which notifications to receive via email and/or in-app:
- • New followers
- • Comments on your posts
- • Likes and shares
- • Replies to your comments
- • Posts from followed authors
- • Newsletter and digest emails
- Set your preferred notification frequency (immediate, daily digest, weekly digest).
- Click "Save Preferences" to apply your settings.

Troubleshooting

If you encounter any issues while using the application:

- Clear your browser cache and cookies.
- Try using a different supported browser.
- Check your internet connection.
- Ensure you're using the latest version of the application.
- For login issues, use the "Forgot Password" feature to reset your credentials.
- Contact support at support@blogapp.com or use the in-app chat support.

Common Issues and Solutions

Issue: Unable to log in
Solutions:

- Ensure your username/email and password are correct
- Check if Caps Lock is enabled
- Reset your password if you've forgotten it

- Clear browser cookies and try again
- Check if your account has been verified (new accounts)

Issue: Images not uploading
Solutions:
- Check that your image is in a supported format (JPG, PNG, GIF, WebP)
- Ensure the file size is under 5MB
- Try compressing the image first
- Check your internet connection
- Try a different browser or device

Component Documentation

Version: 0.1.0
Generated: 7/22/2025

Component Overview

This document provides detailed information about the key components used in the Blog Web Application. The application follows a component-based architecture with atomic design principles, organizing components into atoms, molecules, organisms, templates, and pages.

Component Hierarchy Diagram:

Component Organization

Components are organized following atomic design principles:

- Atoms: Basic building blocks like Button, Input, Icon, Typography
- Molecules: Simple combinations of atoms like SearchBar, FormField, Card
- Organisms: Complex UI sections like Header, Sidebar, CommentSection
- Templates: Page layouts that arrange organisms into a complete page structure
- Pages: Specific instances of templates that present actual content

Authentication Components

Login Component

Path: src/components/AccountIssue/Login/index.jsx
Purpose: Handles user login functionality with form validation and error handling
Key Features:

- Supports email/username and password authentication
- Integrates with OAuth providers (Google, Apple, Twitter)
- Implements client-side validation before submission
- Handles and displays server errors appropriately
- Supports "Remember me" functionality
- Includes password visibility toggle

```
  redirectPath="/dashboard"
  onLoginSuccess={() => trackAnalytics("user_login")}
  showRememberMe={true}
  showSocialLogin={true}
/>
```

SignUp Component

Path: src/components/AccountIssue/SignUp/index.jsx
Purpose: Handles new user registration with multi-step form process
Key Features:

- Multi-step registration flow (account details, profile setup, preferences)
- Progressive form validation at each step
- Password strength indicator
- Username availability checker
- Terms & conditions acceptance
- Email verification integration
- Profile setup guidance

Props:

```
{
  onSignupComplete: PropTypes.func, // Callback when signup is successful
  referralCode: PropTypes.string, // Optional referral code
  initialStep: PropTypes.number, // Which step to start on (default: 1)
  analyticsTracker: PropTypes.func // Function to track signup steps
}
```

State:

```
{
  currentStep: number,
  formValues: {
    username: string,
    email: string,
    password: string,
    confirmPassword: string,
    firstName: string,
    lastName: string,
    termsAccepted: boolean
  },
  formErrors: { ... }, // Validation errors
  isSubmitting: boolean,
  isUsernameTaken: boolean,
  passwordStrength: number, // 0-5 scale
  serverError: string
}
```

Animation Components

Aurora Component

Path: src/Animations/Aurora/Aurora.jsx
Purpose: Provides a dynamic Aurora Borealis effect as a background visual
Implementation: Uses Canvas API with WebGL for performant particle animations
Key Features:

- Responsive design that adapts to container dimensions
- Configurable colors, intensity, and animation speed
- Performance optimized with requestAnimationFrame
- Automatic pause when not in viewport to save resources
- Fallback static gradient for devices with WebGL disabled

Props:

```
{
  width: PropTypes.oneOfType([PropTypes.number, PropTypes.string]), // Container
width
  height: PropTypes.oneOfType([PropTypes.number, PropTypes.string]), // Container
height
  colorPalette: PropTypes.arrayOf(PropTypes.string), // Array of color hex codes
  particleCount: PropTypes.number, // Number of particles to render
  speed: PropTypes.number, // Animation speed multiplier
```

```
  intensity: PropTypes.number, // Color intensity (0.0-1.0)
  interactive: PropTypes.bool, // Whether mouse movement affects animation
  disableAnimation: PropTypes.bool // For performance-sensitive environments
}
```
Usage Example:
```
<Aurora
  width="100%"
  height="400px"
  colorPalette={["#1a2a6c", "#b21f1f", "#fdbb2d"]}
  particleCount={1000}
  speed={1.5}
  intensity={0.8}
  interactive={true}
/>
```

Iridescence Component

Path: src/Animations/Iridescence/Iridescence.jsx
Purpose: Creates a subtle iridescent shimmer effect on UI elements
Implementation: CSS-based animation using gradient overlays and CSS variables
Key Features:

- Light-weight CSS-only implementation for optimal performance
- Configurable gradient direction and shimmer speed
- Supports both light and dark mode with appropriate contrast
- Can be applied to buttons, cards, and section backgrounds
- Accessibility-friendly with reduced motion preference support

Props:
```
{
  width: PropTypes.oneOfType([PropTypes.number, PropTypes.string]), // Container
width
  height: PropTypes.oneOfType([PropTypes.number, PropTypes.string]), // Container
height
  direction: PropTypes.oneOf(['horizontal', 'vertical', 'diagonal']), // Shimmer
direction
  speed: PropTypes.oneOf(['slow', 'medium', 'fast']), // Animation speed
  intensity: PropTypes.number, // Effect intensity (0.0-1.0)
  children: PropTypes.node // Content to apply effect to
}
```

Content Components

BlogPostEditor Component

Path: src/components/Contents/TextEditor/index.jsx
Purpose: Rich text editor for creating and editing blog posts
Implementation: Built on top of TipTap editor with custom extensions
Key Features:

- WYSIWYG editing experience with formatting toolbar
- Support for images, videos, code blocks, and embeds
- Markdown shortcuts for power users (# for headings, ** for bold, etc.)
- Auto-save functionality to prevent data loss
- Draft versioning with restore capability
- Word count and reading time estimation
- SEO optimization suggestions
- Custom block components (callouts, tables, galleries)

Props:
```
{
  initialContent: PropTypes.string, // HTML or markdown content to initialize
editor
  onSave: PropTypes.func.isRequired, // Callback when content is saved
  onPublish: PropTypes.func, // Callback when publish button is clicked
  autoSaveInterval: PropTypes.number, // Time in ms between auto-saves
  toolbarConfig: PropTypes.shape({ // Configure which toolbar items to show
    basic: PropTypes.bool, // bold, italic, underline
    headings: PropTypes.bool,
    lists: PropTypes.bool,
```

```
    media: PropTypes.bool,
    advanced: PropTypes.bool // code, tables, etc.
  }),
  maxLength: PropTypes.number, // Maximum character count
  readOnly: PropTypes.bool, // Whether editor is in read-only mode
  placeholder: PropTypes.string, // Placeholder text when editor is empty
}
```

BlogPostCard Component

Path: src/components/Contents/Item/Article/index.jsx
Purpose: Displays a preview card for blog posts in listings and search results
Implementation: Responsive card component with multiple display variants
Variants:

- Standard: Image, title, excerpt, author, timestamp, engagement metrics
- Compact: Title, author, timestamp only (for sidebar listings)
- Featured: Larger image, title, excerpt, author with background image
- List: Horizontal layout for search results and feed views
- Minimal: Just title and timestamp for notification contexts

Props:

```
{
  post: PropTypes.shape({ // Post data object
    id: PropTypes.string.isRequired,
    title: PropTypes.string.isRequired,
    excerpt: PropTypes.string,
    coverImage: PropTypes.string,
    createdAt: PropTypes.string,
    readTime: PropTypes.number,
    category: PropTypes.string,
    tags: PropTypes.arrayOf(PropTypes.string),
    author: PropTypes.shape({
      id: PropTypes.string,
      name: PropTypes.string,
      avatar: PropTypes.string
    }),
    metrics: PropTypes.shape({
      likes: PropTypes.number,
      comments: PropTypes.number,
      shares: PropTypes.number
    })
  }).isRequired,
  variant: PropTypes.oneOf(['standard', 'compact', 'featured', 'list',
'minimal']),
  onClick: PropTypes.func, // Click handler for the card
  showAuthor: PropTypes.bool, // Whether to show author info
  showMetrics: PropTypes.bool, // Whether to show engagement metrics
  isBookmarked: PropTypes.bool, // Whether post is bookmarked by user
  onBookmarkToggle: PropTypes.func // Handler for bookmark toggle
}
```

Layout Components

AppLayout Component

Path: src/components/Layout/AppLayout.jsx
Purpose: Main application layout wrapper that provides common UI elements
Features:

- Responsive layout with mobile, tablet, and desktop breakpoints
- Header with navigation, search, and user menu
- Optional sidebar with customizable content
- Footer with site links and information
- Toast notification container
- Theme switching capability
- Authentication state awareness

Props:

```
{
```

```
  children: PropTypes.node.isRequired, // Page content
  title: PropTypes.string, // Page title for SEO and browser tab
  description: PropTypes.string, // Meta description
  showHeader: PropTypes.bool, // Whether to show header
  showFooter: PropTypes.bool, // Whether to show footer
  showSidebar: PropTypes.bool, // Whether to show sidebar
  sidebarContent: PropTypes.node, // Custom sidebar content
  headerTransparent: PropTypes.bool, // Whether header has transparent background
  containerClassName: PropTypes.string, // Additional CSS class for container
  requireAuth: PropTypes.bool, // Whether page requires authentication
  fullWidth: PropTypes.bool // Whether to use full width layout
}
```

Deployment Guide

Version: 0.1.0
Generated: 7/22/2025

Deployment Overview

This document provides detailed instructions for deploying the Blog Web Application to various environments. The application is designed to be deployed as a static site that connects to backend API services.

Prerequisites

Before deploying the application, ensure you have:

- Node.js v16.14.0 or higher installed on your build environment
- npm v8.3.0 or higher or yarn v1.22.0 or higher
- Access to the target deployment environment (hosting service credentials)
- Backend API services properly configured and accessible
- Environment variables for the target environment
- SSL certificate for production deployments (required for PWA features)
- Domain name configured with DNS settings pointing to your hosting provider

Environment Configuration

The application requires specific environment variables for each deployment environment:

Production Environment Variables

```
# API Configuration
NEXT_PUBLIC_API_URL=https://api.blogapp.com/v1
NEXT_PUBLIC_SOCKET_URL=https://api.blogapp.com
NEXT_PUBLIC_ASSET_URL=https://cdn.blogapp.com

# Analytics and Monitoring
NEXT_PUBLIC_GOOGLE_ANALYTICS_ID=UA-XXXXXXXXX-X
NEXT_PUBLIC_SENTRY_DSN=https://xxxxxxx.ingest.sentry.io/xxxxxxx

# Authentication
NEXT_PUBLIC_GOOGLE_CLIENT_ID=production-google-client-
id.apps.googleusercontent.com
NEXT_PUBLIC_APPLE_CLIENT_ID=com.blogapp.production

# Feature Flags
NEXT_PUBLIC_ENABLE_EXPERIMENTAL_FEATURES=false
NEXT_PUBLIC_ENABLE_MOCK_API=false
NEXT_PUBLIC_MAINTENANCE_MODE=false
```

Staging Environment Variables

```
# API Configuration
NEXT_PUBLIC_API_URL=https://api-staging.blogapp.com/v1
NEXT_PUBLIC_SOCKET_URL=https://api-staging.blogapp.com
NEXT_PUBLIC_ASSET_URL=https://cdn-staging.blogapp.com

# Analytics and Monitoring
NEXT_PUBLIC_GOOGLE_ANALYTICS_ID=UA-XXXXXXXX-X
NEXT_PUBLIC_SENTRY_DSN=https://xxxxxxx.ingest.sentry.io/staging-xxxxxxx

# Authentication
NEXT_PUBLIC_GOOGLE_CLIENT_ID=staging-google-client-id.apps.googleusercontent.com
NEXT_PUBLIC_APPLE_CLIENT_ID=com.blogapp.staging
```

```
# Feature Flags
NEXT_PUBLIC_ENABLE_EXPERIMENTAL_FEATURES=true
NEXT_PUBLIC_ENABLE_MOCK_API=false
NEXT_PUBLIC_MAINTENANCE_MODE=false
```

Building for Production

To create an optimized production build, follow these steps:

- Ensure all environment variables are properly configured in your .env.production file
- Run linting and tests to ensure code quality: npm run lint && npm test
- Generate the production build: npm run build
- Verify the build by running it locally: npm run start
- The optimized production files will be created in the "build" directory

```
# Example build script for CI/CD pipeline
npm ci --production
npm run lint
npm test
npm run build
```

The production build includes:

- Minified JavaScript bundles with code splitting
- Optimized CSS with vendor prefixes
- Compressed static assets
- Service worker for offline functionality
- Web App Manifest for PWA capabilities
- Static HTML for initial rendering
- Source maps for error tracking (optional, can be disabled)

Deployment Options

Deploying to Netlify

Netlify provides an easy way to deploy the application with continuous integration:

- Create a netlify.toml file in the project root:

```
[build]
  command = "npm run build"
  publish = "build"

[context.production]
  environment = { NODE_VERSION = "16.14.0" }

[[redirects]]
  from = "/*"
  to = "/index.html"
  status = 200
```

- Connect your GitHub repository to Netlify:
- 1. Log in to Netlify and click "New site from Git"
- 2. Select GitHub and authorize Netlify
- 3. Select your repository
- 4. Configure build settings using the netlify.toml file
- 5. Configure environment variables in the Netlify UI (Settings > Build & deploy > Environment)
- 6. Click "Deploy site"
- Set up a custom domain:
- 1. Go to Domain settings in Netlify
- 2. Click "Add custom domain"

- 3. Enter your domain and follow instructions to configure DNS

Deploying to Vercel

Vercel is optimized for Next.js applications and offers similar ease of use:

- Create a vercel.json file in the project root:

```
{
  "version": 2,
  "builds": [
    { "src": "package.json", "use": "@vercel/static-build" }
  ],
  "routes": [
    { "src": "/static/(.*)", "dest": "/static/$1" },
    { "src": "/favicon.ico", "dest": "/favicon.ico" },
    { "src": "/asset-manifest.json", "dest": "/asset-manifest.json" },
    { "src": "/manifest.json", "dest": "/manifest.json" },
    { "src": "/service-worker.js", "headers": { "cache-control": "s-maxage=0" },
"dest": "/service-worker.js" },
    { "src": "/(.*)", "dest": "/index.html" }
  ]
}
```

- Deploy to Vercel:
- 1. Install Vercel CLI: npm i -g vercel
- 2. Login to Vercel: vercel login
- 3. Deploy: vercel --prod
- Alternatively, connect your GitHub repository to Vercel:
- 1. Log in to Vercel and click "New Project"
- 2. Select your repository
- 3. Configure build settings and environment variables
- 4. Click "Deploy"

Deploying to AWS S3 + CloudFront

For more control and scalability, deploy to AWS using S3 for storage and CloudFront for content delivery:

- Create an S3 bucket:
- 1. Go to AWS S3 console and create a new bucket
- 2. Enable "Static website hosting" in bucket properties
- 3. Set the index document to "index.html" and error document to "index.html"
- 4. Update the bucket policy to allow public read access:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "PublicReadGetObject",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::your-bucket-name/*"
    }
  ]
}
```

- Deploy the build to S3:
- 1. Build the project: npm run build
- 2. Deploy to S3 using AWS CLI:

```
aws s3 sync build/ s3://your-bucket-name --delete
```

- Set up CloudFront:

- 1. Create a new CloudFront distribution
- 2. Set the origin domain to your S3 website endpoint
- 3. Configure cache behavior:
- • Default TTL: 86400 (1 day)
- • Compress objects automatically: Yes
- • Forward query strings: Yes
- 4. Set the default root object to "index.html"
- 5. Configure error pages:
- • HTTP Error Code: 403, 404
- • Response Page Path: /index.html
- • HTTP Response Code: 200
- 6. Create and attach SSL certificate using AWS Certificate Manager
- 7. Set up custom domain in CloudFront settings

Deploying with Docker

For containerized deployments, use Docker with Nginx as a web server:

- Create a Dockerfile in the project root:

```
# Build stage
FROM node:16-alpine as build
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Production stage
FROM nginx:alpine
COPY --from=build /app/build /usr/share/nginx/html
COPY nginx/nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Create an nginx.conf file in an nginx directory:

```
server {
  listen 80;
  server_name _;
  root /usr/share/nginx/html;
  index index.html;

  # Enable gzip
  gzip on;
  gzip_types text/plain text/css application/json application/javascript text/xml
application/xml application/xml+rss text/javascript;

  location / {
    try_files $uri $uri/ /index.html;
  }

  # Cache static assets
  location /static/ {
    expires 1y;
    add_header Cache-Control "public, max-age=31536000";
  }
}
```

- Build and run the Docker container:
- 1. Build the Docker image: docker build -t blogapp .
- 2. Run the container: docker run -p 8080:80 blogapp
- 3. Access the application at http://localhost:8080
- For production deployment:
- 1. Push the image to a container registry (Docker Hub, AWS ECR, etc.)

- 2. Deploy to your container orchestration platform (Kubernetes, ECS, etc.)

Continuous Integration/Continuous Deployment

Set up CI/CD pipelines to automate testing and deployment:

GitHub Actions

Create a workflow file at .github/workflows/deploy.yml:

```
name: Deploy

on:
  push:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: 16
          cache: 'npm'
      - name: Install dependencies
        run: npm ci
      - name: Run linter
        run: npm run lint
      - name: Run tests
        run: npm test

  deploy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: 16
          cache: 'npm'
      - name: Install dependencies
        run: npm ci
      - name: Build
        run: npm run build
        env:
          NEXT_PUBLIC_API_URL: ${{ secrets.NEXT_PUBLIC_API_URL }}
          # Add other environment variables
      - name: Deploy to AWS S3
        uses: jakejarvis/s3-sync-action@master
        with:
          args: --delete
        env:
          AWS_S3_BUCKET: ${{ secrets.AWS_S3_BUCKET }}
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          SOURCE_DIR: "build"
      - name: Invalidate CloudFront cache
        uses: chetan/invalidate-cloudfront-action@master
        env:
          DISTRIBUTION: ${{ secrets.CLOUDFRONT_DISTRIBUTION_ID }}
          PATHS: "/*"
          AWS_REGION: "us-east-1"
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

Post-Deployment Verification

After deploying, perform these checks to ensure everything is working correctly:

- Verify all pages load correctly and are responsive on different devices
- Check that API endpoints are connected and returning data
- Test authentication flows (login, signup, password reset)
- Ensure static assets (images, fonts, etc.) are loading properly
- Verify service worker registration and offline functionality
- Run Lighthouse audit to check performance, accessibility, SEO, and PWA compliance
- Check console for any errors or warnings
- Monitor error tracking service (Sentry) for any unexpected errors
- Test critical user flows (posting, commenting, navigation)

Rollback Procedures

If issues are detected after deployment, follow these rollback procedures:

Netlify/Vercel Rollback

- **Go to the Deployments section in your hosting dashboard**
- **Find the last known good deployment**
- **Click "Restore deployment" or equivalent option**
- **Verify the rollback was successful**

AWS S3/CloudFront Rollback

- **Deploy the previous version from CI/CD or manually:**
- **aws s3 sync s3://your-backup-bucket/previous-version/ s3://your-production-bucket/ --delete**
- **Invalidate CloudFront cache:**
- **aws cloudfront create-invalidation --distribution-id YOUR_DISTRIBUTION_ID --paths "/*"**

Development Guide

Version: 0.1.0
Generated: 7/22/2025

Development Overview

This document provides comprehensive guidelines and best practices for developing the Blog Web Application. It covers setup instructions, architecture, coding standards, and workflows for contributors.

Development Environment Setup

To set up your development environment, follow these steps:

Prerequisites

- **Node.js v16.14.0 or higher (LTS recommended)**
- **npm v8.3.0 or higher (comes with Node.js)**
- **Git v2.30.0 or higher**
- **Visual Studio Code (recommended) with ESLint and Prettier extensions**
- **Chrome or Firefox with React Developer Tools extension**

Installation Steps

- **Clone the repository: git clone https://github.com/organization/blog-web-app.git**
- **Navigate to the project directory: cd blog-web-app**
- **Install dependencies: npm install**
- **Create a .env.local file based on .env.example and configure environment variables**
- **Start the development server: npm run dev**
- **The application will be available at http://localhost:3000**

Environment Variables

Create a .env.local file in the project root with the following variables:

```
NEXT_PUBLIC_API_URL=http://localhost:4000/api
NEXT_PUBLIC_SOCKET_URL=http://localhost:4000
NEXT_PUBLIC_ASSET_URL=http://localhost:4000/assets
NEXT_PUBLIC_GOOGLE_ANALYTICS_ID=UA-XXXXXXXXX-X
NEXT_PUBLIC_GOOGLE_CLIENT_ID=your-google-client-id.apps.googleusercontent.com
NEXT_PUBLIC_ENABLE_MOCK_API=true # Set to false when using real API
```

Project Structure

The project follows a feature-based organization with the following structure:

- src/ - Source code directory
- % % % Animations/ - Animation components and effects
- % % % components/ - React components organized by feature
- % % % % AccountIssue/ - Authentication components
- % % % % Contents/ - Content display components
- % % % % Errors/ - Error handling and display components
- % % % % Header/ - Navigation and header components
- % % % % redux/ - Redux store, slices, and actions
- % % % % ... - Other component categories
- % % % Providers/ - React context providers
- % % % % NotificationProvider.jsx - Notification system provider

- % % % % SearchProvider.js - Search context provider
- % % % Themes/ - Theme definitions and theming context
- % % % util/ - Utility functions and helpers
- % % % % data_structures/ - Custom data structure implementations
- % % % % io_utils/ - API and I/O utilities
- % % % App.js - Main application component
- % % % index.js - Application entry point
- public/ - Static assets and files
- % % % images/ - Image assets
- % % % serviceworkers/ - Service worker scripts
- % % % webworkers/ - Web worker scripts

Feature Organization

Each feature is organized following this structure:

- components/FeatureName/ - Root feature directory
- % % % index.jsx - Main entry point/container component
- % % % FeatureName.module.css - Feature-specific styles (if not using styled components)
- % % % SubComponent/ - Sub-component directory
- % % % % index.jsx - Sub-component implementation
- % % % % SubComponent.test.jsx - Component tests
- % % % hooks/ - Feature-specific custom hooks
- % % % utils/ - Feature-specific utility functions

Development Workflow

Follow these guidelines when developing new features or fixing bugs:

Git Workflow

- **Always branch from develop: git checkout -b feature/your-feature-name**
- **Use conventional commit messages: feat(component): add new feature**
- **Submit pull requests against the develop branch**
- **Ensure CI checks pass before requesting review**
- **Squash commits before merging**

Branch Naming Convention

- **feature/feature-name - For new features**
- **bugfix/issue-description - For bug fixes**
- **hotfix/issue-description - For critical production fixes**
- **refactor/component-name - For code refactoring**
- **docs/documentation-description - For documentation updates**

Commit Message Format

Follow the Conventional Commits specification:

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

Types include:

- feat: A new feature
- fix: A bug fix
- docs: Documentation changes
- style: Changes that do not affect the meaning of the code
- refactor: Code change that neither fixes a bug nor adds a feature
- perf: Code change that improves performance

- test: Adding or correcting tests
- build: Changes to the build system or external dependencies
- ci: Changes to CI configuration files and scripts

Coding Standards

Adhere to the following coding standards to maintain code quality:

General Guidelines

- **Use functional components with hooks instead of class components**
- **Use TypeScript for type safety when adding new components**
- **Follow the principle of single responsibility (each component does one thing well)**
- **Keep components small and focused (< 300 lines recommended)**
- **Use named exports instead of default exports for better refactoring support**
- **Avoid prop drilling; use context or state management when props are passed through many layers**
- **Prefer composition over inheritance for component reuse**
- **Use error boundaries to gracefully handle component errors**

React Best Practices

- **Memoize expensive components using React.memo()**
- **Use the useCallback() hook for event handlers passed to child components**
- **Use the useMemo() hook for expensive calculations**
- **Lazy load components for code splitting: const Component = React.lazy(() => import('./Component'))**
- **Add meaningful alt text to all images for accessibility**
- **Use semantic HTML elements (e.g., <button> instead of)**
- **Use proper ARIA attributes for accessibility**
- **Avoid direct DOM manipulation; use refs when necessary**

State Management

- **Use local state (useState) for component-specific state**
- **Use context (useContext) for state shared between a few components**
- **Use Redux for application-wide state or complex state logic**
- **Follow the Redux Toolkit guidelines for reducer organization**
- **Use createSlice() to define reducers and actions**
- **Normalize complex state structures in Redux**
- **Use selectors for accessing and computing derived state**

Testing

The project uses Jest and React Testing Library for testing. Run tests with:

```
npm test              # Run all tests
npm test -- --watch  # Run tests in watch mode
```

Follow these guidelines for writing tests:

Component Testing

- **Test behavior rather than implementation details**
- **Use React Testing Library's queries in this order of preference:**
- **1. Accessible queries (getByRole, getByLabelText, getByPlaceholderText, getByText)**
- **2. Test ID queries (getByTestId) as a last resort**
- **Simulate user interactions using userEvent rather than fireEvent when possible**

- **Test alternative states (loading, error, empty, etc.)**
- **Use mock data consistently across tests**
- **Isolate component tests by mocking external dependencies**

Test Organization

- **Co-locate tests with the components they test**
- **Use descriptive test names following the pattern: "renders/behaves/handles [expected behavior] when [condition]"**
- **Group related tests using describe blocks**
- **Use beforeEach for common setup logic**
- **Keep tests independent of each other**

Example Test

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import LoginForm from './LoginForm';

describe('LoginForm', () => {
  const mockLogin = jest.fn();

  beforeEach(() => {
    mockLogin.mockClear();
  });

  it('submits username and password when form is submitted', async () => {
    render(<LoginForm onSubmit={mockLogin} />);

    // Fill out form
    await userEvent.type(screen.getByLabelText(/username/i), 'testuser');
    await userEvent.type(screen.getByLabelText(/password/i), 'password123');

    // Submit form
    await userEvent.click(screen.getByRole('button', { name: /log in/i }));

    // Assert
    expect(mockLogin).toHaveBeenCalledWith({
      username: 'testuser',
      password: 'password123',
    });
  });
});
```

Performance Optimization

Follow these guidelines to ensure optimal application performance:

- Use React.lazy() and Suspense for code splitting
- Implement windowing for long lists with react-window or react-virtualized
- Optimize images using WebP format and responsive sizes
- Use pagination or infinite scrolling for large datasets
- Implement proper caching strategies for API calls
- Memoize expensive calculations with useMemo()
- Use service workers for offline support and caching
- Minimize bundle size by tree-shaking and dynamic imports
- Optimize critical rendering path by deferring non-essential resources

Performance Monitoring

Use the following tools to monitor and improve performance:

- **Lighthouse in Chrome DevTools for overall performance audits**
- **React DevTools Profiler for component rendering performance**
- **WebPageTest for real-world performance testing**

- **Bundle analyzer: npm run analyze**

Troubleshooting

Common development issues and solutions:

Node Module Issues

If you encounter dependency issues:

- **Delete node_modules directory: rm -rf node_modules**
- **Clear npm cache: npm cache clean --force**
- **Reinstall dependencies: npm install**

API Connection Issues

If the application cannot connect to the API:

- **Ensure the API server is running**
- **Check that NEXT_PUBLIC_API_URL is correctly set in .env.local**
- **Verify network connectivity and CORS configuration**
- **Check browser console for specific error messages**
- **Set NEXT_PUBLIC_ENABLE_MOCK_API=true to use mock data for development**

Architecture Documentation

Version: 0.1.0
Generated: 7/22/2025

Architecture Overview

This document provides a comprehensive description of the Vydeo application architecture. The application is built as a modern React single-page application (SPA) with a focus on real-time communication, video content management, and social features.

High-Level Architecture

The Vydeo application follows a client-side architecture built with React 18, using Redux for global state management and Context API for theme and UI state. The application communicates with multiple backend services via RESTful APIs and uses WebSockets for real-time features including chat, notifications, and content updates.

The application architecture consists of the following major layers:

- Presentation Layer: React components with Material UI, organized by feature
- State Management Layer: Redux store for global state and Context API for theme and UI state
- Service Layer: API clients, WebSocket communication, and utility services
- Persistence Layer: LocalForage and browser storage for offline data access

Data Flow Diagram:

System Context

The Vydeo application interacts with several external systems:

- Java API Service (https://apis.vydeo.xyz/java): Primary RESTful backend for core functionality
- Python API Service (https://apis.vydeo.xyz/py): Specialized service for data processing

```
    const token = localStorage.getItem("token");
    if (token) {
      config.headers.token = token;
    }
    return config;
  }
);
```

Core Components

Frontend Architecture

The frontend application is organized into feature-based components with the following structure:

- src/ - Source code directory
- % % % components/ - React components organized by feature
- % % % % AccountIssue/ - Authentication related components
- % % % % % Login/ - Login functionality
- % % % % % SignUp/ - Registration functionality
- % % % % % Forget/ - Password recovery
- % % % % Contents/ - Main content components
- % % % % % Videos/ - Video browsing and playback
- % % % % % Chat/ - Messaging functionality
- % % % % % FriendList/ - Friend management
- % % % % % VideoList/ - Video listings and recommendations
- % % % % % Settings/ - User preferences
- % % % % Header/ - Navigation and search components
- % % % % Errors/ - Error handling components
- % % % % redux/ - Redux state management
- % % % Providers/ - Context providers
- % % % % NotificationProvider.jsx - Notification handling
- % % % % SearchProvider.js - Search functionality
- % % % Themes/ - Theme definitions
- % % % % ThemeContext.js - Light/dark mode theming
- % % % Animations/ - Animation components
- % % % util/ - Utility functions
- % % % % io_utils/ - API and data utilities
- % % % % data_structures/ - Custom data handling
- % % % App.js - Main application component
- % % % index.js - Application entry point

Data Flow

The application follows a unidirectional data flow pattern:

- 1. User interacts with a component, triggering an event handler
- 2. Event handler dispatches an action to the Redux store
- 3. For API operations:
- a. The appropriate API client (apiClient, flaskClient) sends the request
- b. Token interceptors automatically add authentication headers
- c. Response interceptors handle token expiration and errors
- 4. Redux reducers process actions and update state
- 5. Connected components re-render based on state changes
- 6. For real-time features, WebSockets provide immediate updates

Redux Store Structure

```
// Actual Redux store configuration from the codebase
export default configureStore({
  reducer: {
    searchResult: searchResult,
    userDetails: userDetails,
    search: Search,
    refreshMessages: refreshMessagesReducer,
    refreshSideBar: refreshSideBarReducer,
    refreshMailBox: refreshMailBoxReducer,
    auth: authReducer,
    scrollCursor: scrollCursorReducer
  },
})
```

Authentication Flow

The application uses JWT-based authentication with token persistence:

Login Process

- **1. User enters credentials in the Login component**
- **2. On form submission, credentials are sent to the authentication API**
- **3. Server validates credentials and returns a JWT token**
- **4. The Redux login action is dispatched with the token and user data**
- **5. Token is stored in localStorage and user state in Redux**
- **6. API client interceptors automatically use the token for subsequent requests**
- **7. User is redirected to the main application content**

Token Expiration Handling

- **1. API client interceptors detect 401 Unauthorized responses**
- **2. The logout action is dispatched to clear authentication state**
- **3. User is presented with the login screen**
- **4. localStorage and localForage data are cleared for security**

Authentication Code Implementation

```
// Authentication slice from actual codebase
const authSlice = createSlice({
  name: 'auth',
  initialState: {
    isAuthenticated: false,
    showLoginModal: false,
    user: null,
    token: null,
    isLoading: false,
  },
  reducers: {
    login: (state, action) => {
      state.isAuthenticated = true;
      state.user = action.payload.user;
      state.token = action.payload.token;

      // Store auth data in localStorage
      localStorage.setItem('isLoggedIn', 'true');
      localStorage.setItem('token', action.payload.token);
      if (action.payload.user?.userId) {
        localStorage.setItem('userId', action.payload.user.userId);
      }
    },
    logout: (state) => {
      state.isAuthenticated = false;
```

```
      state.user = null;
      state.token = null;

      // Clear stored auth data
      localStorage.removeItem('isLoggedIn');
      localStorage.removeItem('token');
      localStorage.removeItem('userId');
    }
    // ... other reducers
  }
});
```

Theme Management

The application implements a theme switching capability using React Context and Material UI's ThemeProvider:

- 1. ThemeContext.js defines the theme context and provider
- 2. Theme preferences (light/dark) are persisted in localStorage
- 3. The ThemeContextProvider wraps the application and provides theme values
- 4. Material UI theme is dynamically created based on the selected mode
- 5. Component styling is adjusted automatically based on the current theme

```
// Theme implementation from the actual codebase
export const ThemeContextProvider = ({ children }) => {
  const [mode, setMode] = useState('dark');

  // Load saved theme from localStorage
  useEffect(() => {
    const savedMode = localStorage.getItem('mode');
    if (savedMode === 'light' || savedMode === 'dark') {
      setMode(savedMode);
    }
  }, []);

  // Save theme changes to localStorage
  useEffect(() => {
    localStorage.setItem('mode', mode);
  }, [mode]);

  const toggleMode = () => {
    setMode((prev) => (prev === 'light' ? 'dark' : 'light'));
  };

  // Create Material UI theme based on mode
  const theme = useMemo(() => createTheme({
    palette: {
      mode,
      // Custom theme colors
      background: {
        default: mode === 'dark' ? '#000000' : '#ffffff',
        paper: mode === 'dark' ? 'rgba(0,0,0,0.6)' : '#ffffff',
      },
      // ... other theme settings
    }
  }), [mode]);

  return (
    <ThemeContext.Provider value={{ mode, toggleMode }}>
      <ThemeProvider theme={theme}>
        <CssBaseline enableColorScheme />
        {children}
      </ThemeProvider>
    </ThemeContext.Provider>
  );
};
```

Offline Support and Data Persistence

The application implements offline support and data persistence through:

- Service Workers: Registered for background processing and push notifications
- LocalForage: IndexedDB-based persistent storage for application data
- LocalStorage: For simple key-value storage of user preferences and tokens
- Push Notifications: Using the browser's Push API for notifications when the app is closed

```
// Service worker registration from the actual codebase
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/serviceworkers/
NotificationReceiver.js').then(
    registration => {
      navigator.serviceWorker.ready.then(readyRegistration => {
        if (Notification.permission === 'granted') {
          register(readyRegistration); // Register for push notifications
        } else if (Notification.permission === 'default') {
          Notification.requestPermission().then(permission => {
            if (permission === 'granted') {
              register(readyRegistration);
            }
          });
        }
      });
    }
  );
}
```

Routing and Navigation

The application uses React Router v6 for client-side routing with authentication protection:

- BrowserRouter provides the routing context for the application
- Authentication-based conditional rendering controls access to routes
- The Contents component contains all authenticated routes
- When not authenticated, the AccountIssue component is shown instead
- Error components handle specific error conditions like network issues

```
// Main routing structure from App.js
if (!isAuthenticated || showLoginModal) {
  return (
    <ThemeContextProvider>
      <AccountIssue
        loginState={isAuthenticated}
        setLoginState={(newState) => {
          // Login state management
        }}
      />
    </ThemeContextProvider>
  );
}

if (isAuthenticated) {
  return (
    <BrowserRouter>
      <ThemeContextProvider>
        <Contents
          setLogin={async (newState) => {
            // Login/logout handling
          }}
        />
      </ThemeContextProvider>
    </BrowserRouter>
  );
}
```

Error Handling

The application implements comprehensive error handling:

- Dedicated error components for specific error scenarios:
- • NetworkError: Displayed when internet connectivity is lost
- • EndpointNotAvailableError: Shown when API endpoints are unreachable
- • NotFoundError: For 404 errors when navigating to non-existent routes
- API error handling through axios interceptors
- Authentication error handling with automatic logout on token expiration
- Notifications for user feedback on operations

Real-time Communication

The application uses WebSockets for real-time features:

- WebSocket connection to wss://apis.vydeo.xyz/ws
- Real-time chat messaging between users
- Push notifications for new messages and content updates
- Message handlers for different types of real-time events
- Integration with Redux for state updates on WebSocket events

Security Considerations

The application implements the following security measures:

- JWT-based authentication with secure token storage
- Automatic token handling through API client interceptors
- Token expiration management and automatic logout
- Secure WebSocket communication with token authentication
- Environment-specific API endpoints for development and production
- Proper data clearing on logout to prevent information leakage