Blogify MERN Stack Application - Complete Documentation

Table of Contents

Introduction

Blogify is a full-stack **MERN** (*MongoDB*, *Express.js*, *React.js*, *Node.js*) blog application that allows users to read blog posts and administrators to manage content. The application features a modern React frontend with *Tailwind CSS* styling and a robust Node.js backend with MongoDB database.

Key Features:

- **Public Blog Reading**: Users can browse and read published blog posts
- **Admin Panel**: Secure admin interface for managing blogs and comments
- **Image Upload**: Integration with *ImageKit* for optimized image handling
- **AI Integration**: *Gemini AI* for content enhancement
- **Comment System**: Users can comment on blog posts with admin approval
- **Responsive Design**: Mobile-friendly interface using *Tailwind CSS*

Project Structure Overview

```
Blogify/
├── client/                 # Frontend React Application
│   ├── src/
│   │   ├── components/     # Reusable UI components
│   │   ├── pages/         # Page components
│   │   ├── context/       # Global state management
│   │   └── assets/        # Static assets
│   ├── package.json       # Frontend dependencies
│   ├── vite.config.js     # Build configuration
│   └── tailwind.config.js # Styling configuration
└── server/                # Backend Node.js Application
    ├── controllers/       # Business logic
    ├── models/           # Database schemas
    ├── routes/           # API route definitions
    ├── middleware/       # Custom middleware
    ├── configs/          # Configuration files
    └── package.json      # Backend dependencies
```

Frontend Architecture

1. Entry Point and Configuration

`client/package.json`

```json
{
  "name": "client",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint .",
    "preview": "vite preview"
  }
}
```

Purpose: Defines project metadata, scripts, and dependencies

- **Scripts**: Development server, build process, linting
- **Dependencies**: React, React Router, Axios, *Tailwind CSS*, etc.

`client/vite.config.js`

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
})
```

Purpose: Configures the Vite build tool

- **React Plugin**: Enables JSX transformation and React features
- **Fast Development**: Hot module replacement for quick development

`client/tailwind.config.js`

```
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Purpose: Configures *Tailwind CSS* utility framework

- **Content**: Specifies files to scan for CSS classes
- **Theme**: Customization options for design system

2. Application Bootstrap

`client/index.html`

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

Purpose: HTML template for the React application

- **Root Element**: Container for React app mounting
- **Module Script**: Loads the main JavaScript entry point

`client/src/main.jsx`

```
import { StrictMode } from 'react'
import { createRoot } from 'react-dom/client'
import './index.css'
import App from './App.jsx'
import { BrowserRouter } from 'react-router-dom'
import { AppProvider } from './context/AppContext.jsx'
import { StyleSheetManager } from 'styled-components'

createRoot(document.getElementById('root')).render(
  <StrictMode>
    <StyleSheetManager shouldForwardProp={() => true}>
      <BrowserRouter>
        <AppProvider>
          <App />
        </AppProvider>
      </BrowserRouter>
    </StyleSheetManager>
  </StrictMode>,
)
```

Purpose: Application entry point and provider setup

- **StrictMode**: Enables additional development checks
- **BrowserRouter**: Enables client-side routing
- **AppProvider**: Provides global state context
- **StyleSheetManager**: Manages styled-components

3. Global State Management

`client/src/context/AppContext.jsx`

```
import { createContext, useContext, useState, useEffect } from 'react';
import axios from 'axios';

const AppContext = createContext();

export const AppProvider = ({ children }) => {
    const [token, setToken] = useState(localStorage.getItem('token') || '');
    const [blogs, setBlogs] = useState([]);
    const [isLoggedIn, setIsLoggedIn] = useState(false);

    // Configure axios defaults
    useEffect(() => {
        if (token) {
            axios.defaults.headers.common['Authorization'] = `Bearer ${token}
            setIsLoggedIn(true);
        }
    }, [token]);

    const fetchBlogs = async () => {
        try {
            const response = await axios.get(`${import.meta.env.VITE_BACKEND_
            if (response.data.success) {
                setBlogs(response.data.blogs);
            }
        } catch (error) {
            console.error('Error fetching blogs:', error);
        }
    };

    const login = async (email, password) => {
        try {
            const response = await axios.post(`${import.meta.env.VITE_BACKEND
                email,
                password
            });

            if (response.data.success) {
                const newToken = response.data.token;
                setToken(newToken);
                localStorage.setItem('token', newToken);
                axios.defaults.headers.common['Authorization'] = `Bearer ${ne
                setIsLoggedIn(true);
                return { success: true };
            }
        } catch (error) {
            return { success: false, message: error.response?.data?.message |
        }
    };

    const logout = () => {
        setToken('');
        localStorage.removeItem('token');
        delete axios.defaults.headers.common['Authorization'];
        setIsLoggedIn(false);
    };

    const addBlog = async (blogData) => {
        try {
            const response = await axios.post(`${import.meta.env.VITE_BACKEND
            if (response.data.success) {
                await fetchBlogs(); // Refresh blogs list
                return { success: true };
            }
        } catch (error) {
            return { success: false, message: error.response?.data?.message |
        }
    };

    const value = {
        token,
        blogs,
        isLoggedIn,
        fetchBlogs,
        login,
        logout,
        addBlog,
        setBlogs
    };

    return (
        <AppContext.Provider value={value}>
            {children}
        </AppContext.Provider>
    );
};

export const useAppContext = () => {
    const context = useContext(AppContext);
    if (!context) {
        throw new Error('useAppContext must be used within AppProvider');
    }
    return context;
};
```

Purpose: Centralized state management for the entire application

- **Authentication State**: Manages login/logout and token storage
- **Blog Data**: Handles blog fetching and caching
- **API Configuration**: Sets up Axios with authentication headers
- **Context Provider**: Makes state available to all components

4. Main Application Component

`client/src/App.jsx`

```
import React, { useEffect, useState } from 'react'
import { Routes, Route } from 'react-router-dom'
import { ToastContainer } from 'react-toastify'
import 'react-toastify/dist/ReactToastify.css'
import axios from 'axios'

// Page imports
import Home from './pages/Home'
import Blog from './pages/Blog'
import AdminLogin from './pages/admin/AdminLogin'
import Layout from './pages/admin/Layout'
import AddBlog from './pages/admin/AddBlog'
import ListBlog from './pages/admin/ListBlog'
import Dashboard from './pages/admin/Dashboard'
import Comments from './pages/admin/Comments'

// Context
import { AppProvider, useAppContext } from './context/AppContext'

const App = () => {
  const [isAuthenticated, setIsAuthenticated] = useState(false);

  useEffect(() => {
    const token = localStorage.getItem('token');
    if (token) {
      axios.defaults.headers.common['Authorization'] = `Bearer ${token}`;
      setIsAuthenticated(true);
    }
  }, []);

  return (
    <AppProvider>
      <div className="App">
        <ToastContainer
          position="top-right"
          autoClose={3000}
          hideProgressBar={false}
          newestOnTop={false}
          closeOnClick
          rtl={false}
          pauseOnFocusLoss
          draggable
          pauseOnHover
        />

        <Routes>
          <!-- Public Routes -->
          <Route path="/" element={<Home />} />
          <Route path="/blog/:id" element={<Blog />} />

          <!-- Admin Login Route -->
          <Route path="/admin" element={<AdminLogin />} />

          <!-- Protected Admin Routes -->
          <Route path="/admin/*" element={
            <Layout>
              <Routes>
                <Route path="dashboard" element={<Dashboard />} />
                <Route path="add-blog" element={<AddBlog />} />
                <Route path="list-blog" element={<ListBlog />} />
                <Route path="comments" element={<Comments />} />
              </Routes>
            </Layout>
          } />
        </Routes>
      </div>
    </AppProvider>
  )
}

export default App
```

Purpose: Main application component with routing configuration

Route Definition: Maps URLs to components

Authentication Check: Verifies token on app load

Toast Notifications: Global notification system

Layout Wrapper: Admin routes wrapped in layout component

Backend Architecture

1. Server Configuration

`server/package.json`

```
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "type": "module",
  "scripts": {
    "server": "nodemon server.js",
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.21.2",
    "mongoose": "^8.9.3",
    "jsonwebtoken": "^9.0.2",
    "bcrypt": "^5.1.1",
    "multer": "^1.4.5-lts.1",
    "cors": "^2.8.5",
    "dotenv": "^16.4.7",
    "imagekit": "^5.2.0"
  },
  "devDependencies": {
    "nodemon": "^3.1.9"
  }
}
```

Purpose: Backend project configuration and dependencies

Express: Web framework for Node.js

Mongoose: MongoDB object modeling

JWT: Authentication token management

- **Multer**: File upload handling
- **ImageKit**: Image optimization service

`server/server.js`

```
import express from 'express'
import 'dotenv/config'
import cors from 'cors'
import connectDB from './configs/db.js';
import adminRouter from './routes/adminRoutes.js';
import blogRouter from './routes/blogRoutes.js';

const app = express();

// Connect to database
await connectDB()

// Middlewares
app.use(cors())
app.use(express.json())

// Routes
app.get('/', (req, res) => res.send("API is working on port..."))
app.use('/api/admin', adminRouter)
app.use('/api/blog', blogRouter)

const PORT = process.env.PORT || 5000;

app.listen(PORT, () => {
    console.log('server is running on port ' + PORT)
})

export default app;
```

Purpose: Main server entry point

- **Database Connection**: Establishes MongoDB connection
- **Middleware Setup**: CORS and JSON parsing
- **Route Registration**: Maps API endpoints to routers
- **Server Startup**: Listens on specified port

2. Database Configuration

`server/configs/db.js`

```
import mongoose from "mongoose";

const connectDB = async () => {
    try {
        await mongoose.connect(`${process.env.MONGODB_URI}/blogify`);
        console.log("Database Connected");
    } catch (error) {
        console.log("Database Connection Error:", error.message);
        process.exit(1); // Exit process with failure
    }
}

export default connectDB;
```

Purpose: Database connection management

- **MongoDB Connection**: Uses Mongoose to connect to MongoDB
- **Error Handling**: Graceful error handling with process exit
- **Environment Variables**: Uses secure connection string

Database Models

1. Blog Model

`server/models/Blog.js`

```
import mongoose from "mongoose"

const blogSchema = new mongoose.Schema({
    title: {
        type: String,
        required: true
    },
    subtitle: {
        type: String
    },
    description: {
        type: String,
        required: true
    },
    category: {
        type: String,
        required: true
    },
    image: {
        type: String,
        required: true
    },
    isPublished: {
        type: Boolean,
        required: true,
        default: false
    }
}, {timestamps: true});

const Blog = mongoose.model('Blog', blogSchema);
export default Blog;
```

Purpose: Defines the structure for blog posts

- **Required Fields**: Title, description, category, image
- **Optional Fields**: Subtitle
- **Publishing Control**: isPublished flag for draft/published state
- **Timestamps**: Automatic createdAt and updatedAt fields

2. Comment Model

`server/models/Comment.js`

```
import mongoose from "mongoose";

const commentSchema = new mongoose.Schema({
    blog: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'Blog',
        required: true
    },
    name: {
        type: String,
        required: true,
        trim: true
    },
    content: {
        type: String,
        required: true,
        trim: true,
        minlength: 1
    },
    isApproved: {
        type: Boolean,
        required: true,
        default: false
    }
}, {
    timestamps: true,
    versionKey: false
});

const Comment = mongoose.model('Comment', commentSchema);
export default Comment;
```

Purpose: Defines the structure for blog comments

- **Blog Reference**: Links comments to specific blog posts
- **User Information**: Name and content fields
- **Moderation**: isApproved flag for comment approval
- **Data Validation**: Trim whitespace and minimum length

API Endpoints

1. Blog Controller

`server/controllers/blogController.js`

Key Functions:

- **addBlog**: Creates new blog posts with image upload

```
export const addBlog = async (req, res) => {
    try {
        const { title, subtitle, description, category, isPublished } = JSON.
        const imageFile = req.file;

        // Validate required fields
        if (!title || !description || !imageFile) {
            return res.status(400).json({ success: false, message: "Missing r
        }

        // Upload image to ImageKit
        const fileBuffer = fs.readFileSync(imageFile.path);
        const response = await imagekit.upload({
            file: fileBuffer,
            fileName: imageFile.originalname,
            folder: "/blogs"
        });

        // Create optimized image URL
        const optimizedImageUrl = imagekit.url({
            path: response.filePath,
            transformation: [{
                quality: 'auto',
                format: 'webp',
                width: 1280
            }]
        });

        // Save blog to database
        await Blog.create({
            title,
            subtitle,
            description,
            category,
            image: optimizedImageUrl,
            isPublished: isPublished || false
        });

        // Clean up temporary file
        fs.unlinkSync(imageFile.path);

        res.status(200).json({ success: true, message: "Blog added successful
    } catch (error) {
        console.error('Error adding blog:', error);
        res.status(500).json({ success: false, message: error.message });
    }
}
```

getAllBlogs: Retrieves all published blogs

```
export const getAllBlogs = async (req, res) => {
    try {
        const blogs = await Blog.find({isPublished: true});
        res.json({success: true, blogs});
    } catch (error) {
        res.json({success: false, message: error.message});
    }
}
```

getBlogById: Retrieves a specific blog by ID

```
export const getBlogById = async (req, res) => {
    try {
        const {blogId} = req.params;
        const foundBlog = await Blog.findById(blogId);
        if (!foundBlog)
            return res.json({success: false, message: "Blog not found"});
        res.json({success: true, blog: foundBlog});
    } catch (error) {
        res.json({success: false, message: error.message});
    }
}
```

2. Admin Controller

`server/controllers/adminController.js`

Key Functions:

- **loginAdmin**: Handles admin authentication
- **getAllBlogs**: Retrieves all blogs (including unpublished)
- **deleteBlogs**: Removes blogs and associated comments

Frontend-Backend Communication

1. API Configuration

The frontend uses *Axios* for HTTP requests with the following configuration:

```
// In AppContext.jsx
useEffect(() => {
    if (token) {
        axios.defaults.headers.common['Authorization'] = `Bearer ${token}`;
        setIsLoggedIn(true);
    }
}, [token]);
```

2. Data Flow Examples

Blog Fetching Process:

- **Frontend Request**: Component calls `fetchBlogs()` from context
- **API Call**: Axios sends GET request to `/api/blog/list`
- **Backend Processing**: Blog controller queries database
- **Database Query**: MongoDB returns published blogs
- **Response**: Backend sends JSON response
- **Frontend Update**: Context updates blogs state
- **UI Render**: Components re-render with new data

Authentication Flow:

- **Login Form**: User submits credentials
- **API Request**: POST to `/api/admin/login`
- **Backend Validation**: Check credentials against database
- **Token Generation**: JWT token created if valid
- **Response**: Token sent to frontend
- **Storage**: Token stored in localStorage
- **Header Setup**: Axios configured with Authorization header

File Upload Process:

- **File Selection**: User selects image file
- **FormData Creation**: Frontend creates multipart form data
- **Upload Request**: POST to `/api/blog/add` with file
- **Multer Processing**: Backend middleware handles file upload
- **ImageKit Upload**: File uploaded to cloud storage
- **Database Save**: Blog data saved with image URL
- **Cleanup**: Temporary file removed from server

File-by-File Analysis

Frontend Components

`client/src/components/Header.jsx`

- **Purpose**: Main navigation and search functionality
- **Features**: Logo, navigation menu, search bar
- **State Management**: Uses context for authentication state

`client/src/components/BlogCard.jsx`

- **Purpose**: Displays blog preview cards
- **Props**: Blog data (title, image, excerpt)
- **Navigation**: Links to full blog view

`client/src/components/AIButton.jsx`

- **Purpose**: AI-powered content enhancement
- **Integration**: Connects to *Gemini AI* service
- **Functionality**: Content suggestions and improvements

Frontend Pages

`client/src/pages/Home.jsx`

- **Purpose**: Landing page with blog listings
- **Components**: Header, BlogCard grid, pagination
- **Data**: Fetches and displays published blogs

`client/src/pages/Blog.jsx`

- **Purpose**: Individual blog post view
- **Features**: Full content, comments section
- **Dynamic Routing**: Uses URL parameters for blog ID

Admin Pages

`client/src/pages/admin/Dashboard.jsx`

- **Purpose**: Admin overview and statistics
- **Metrics**: Blog count, comment count, recent activity

`client/src/pages/admin/AddBlog.jsx`

- **Purpose**: Blog creation interface
- **Features**: Rich text editor, image upload, category selection
- **Validation**: Form validation and error handling

`client/src/pages/admin/ListBlog.jsx`

- **Purpose**: Blog management interface
- **Features**: Blog list, edit/delete actions, publish toggle

Backend Structure

Controllers Folder Purpose:

- **Separation of Concerns**: Business logic separated from routes
- **Reusability**: Controller functions can be used by multiple routes

- **Testing**: Easier to unit test business logic
- **Maintainability**: Centralized logic for easier updates

Models Folder Purpose:

- **Data Structure**: Defines database schema and validation
- **Consistency**: Ensures data integrity across application
- **Relationships**: Defines connections between collections
- **Validation**: Built-in data validation rules

Routes Folder Purpose:

- **URL Mapping**: Maps HTTP endpoints to controller functions
- **Middleware Integration**: Applies authentication and validation
- **Organization**: Groups related endpoints together
- **RESTful Design**: Follows REST API conventions

Middleware Folder Purpose:

- **Authentication**: Verifies JWT tokens
- **File Upload**: Handles multipart form data
- **Error Handling**: Centralized error processing
- **Request Processing**: Modifies requests before reaching controllers

---

Conclusion

This Blogify application demonstrates a complete **MERN** stack implementation with:

- **Modern Frontend**: React with hooks, context API, and *Tailwind CSS*
- **Robust Backend**: Express.js with proper MVC architecture
- **Secure Authentication**: JWT-based admin authentication
- **File Management**: Cloud-based image storage with optimization
- **Database Design**: Well-structured MongoDB schemas
- **API Design**: RESTful endpoints with proper error handling

The application showcases best practices in full-stack development, including proper separation of concerns, secure authentication, optimized file handling, and responsive design.