DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

CU CHANDIGARH UNIVERSITY

Discover. Learn. Empower.

Report File FULL STACK

Student Name: Souradeep Banerjee

UID: 23BAI70654

Branch: BE-AIT-CSE

Section/Group: 23AIT-KRG-G2

Semester: 5th

Date of Performance: 25th Aug, 2025

Subject Name: Full Stack

Subject Code: 23CSP-339

Project Report: Blogify

A Client-Side Blogging Website with React.js

Executive Summary

This report details the design, development, and functionality of "Blogify," a modern, frontend-only blogging application built with React.js. The primary objective was to create a fully functional blog platform that operates entirely within the user's browser, requiring no backend server or database.

Key features include dynamic page routing, a "New Blog" modal for content creation, and a robust localStorage persistence system. The application's most advanced feature is its ability to directly upload a Microsoft Word (.docx) document, parse its content (including images), and render it as a blog post.

The project successfully demonstrates advanced client-side capabilities while also exploring the inherent limitations of a serverless architecture, particularly the storage quotas of localStorage.

---

1. Introduction

1.1 Project Objective

The main goal was to create a responsive, single-page application (SPA) where a user can write, save, and manage personal blog posts. The project aimed to deliver a seamless user experience comparable to a full-stack application, but using only frontend technologies (HTML, CSS, and React.js).

1.2 Problem Statement

Many users desire a simple, private space to write and preview blog posts without the complexity of setting up a database or a web server. This project provides a "what you see is what you get" (WYSIWYG) solution that persists data locally, making it an ideal digital notebook or a draft space for writers.

1.3 Scope

- **In-Scope:**
  - A "Discover" landing page with mock content.
  - A "My Blogs" page listing all user-created posts.
  - A modal (dialog box) for creating new blogs.
  - Manual text entry for blog posts.
  - A file uploader to parse and render .docx files.
  - In-browser parsing of text and images from Word documents.
  - Dynamic routing to display individual blog posts on a unique page.
  - Data persistence using the browser's localStorage.
  - A modern, responsive UI with a "dark blue, white, and green" theme and the "Lato" font.
- **Out-of-Scope:**
  - User authentication (login/signup).
  - A backend server or cloud database.
  - Multi-user support (all blogs are local to a single browser).
  - Server-side rendering or cloud storage.

---

2. Technology Stack

This project was built exclusively with frontend technologies:

- **React.js (v18+):** The core library for building the user interface. We used React Hooks (like useState and useEffect) for all state management and component lifecycle logic.
- **React Router (react-router-dom):** Used to create a multi-page feel within a single-page application. It handles all navigation (e.g., /, /home, /blog/:blogId) dynamically.
- **HTML5 & CSS3:** Used for the application's structure and styling. The design was customized with a specific color palette and the "Lato" web font, implemented directly in App.css and index.css.
- **mammoth.js:** A critical third-party library. Its sole purpose is to read .docx files (as ArrayBuffer) and convert them into HTML, all within the browser. This is what enables the Word document upload feature.
- **Browser localStorage (Web API):** This browser-based key-value store acts as the project's "database." It allows us to save the user's array of blog posts as a JSON string, which persists even after the browser is closed.

---

3. System Architecture & Design

The application's architecture is centered around the main App.js component, which acts as the "brain" of the operation.

3.1 Component-Based Structure

The UI is broken down into two main categories:

1. **Pages (/pages):** High-level components that are mapped to routes.
   - LandingPage.js: The "Discover" page with static content.
   - HomePage.js: Displays the list of user-created blogs.
   - BlogPostPage.js: A dynamic page that renders a single post.
2. **Components (/components):** Reusable UI elements.

- Navbar.js: Handles navigation.
- BlogCard.js: A card to preview a blog.
- BlogModal.js: The complex form for creating a new post.

3.2 State Management & Data Flow

State is "lifted" to the highest common ancestor, App.js.

1. **Main State:** App.js holds the myBlogs state, which is an array of all blog objects.
2. **Data Loading:** When the app first mounts, a useEffect hook in App.js reads from localStorage and populates the myBlogs state.
3. **Data Saving:** A second useEffect hook "watches" the myBlogs state. Any time this state changes (like when a new blog is added), this effect automatically re-saves the entire array to localStorage.
4. **Prop Drilling:**
   - myBlogs is passed **down** as a prop to HomePage.js (to list the blogs).
   - The addBlog *function* is passed **down** to BlogModal.js, allowing the modal to send a new blog object **up** to App.js and update the main state.

---

4. Key Feature Implementation

4.1 Dynamic Blog Post Routing

This feature allows every blog to have its own unique URL.

1. **Route Definition (in App.js):** A dynamic route is defined: <Route path="/blog/:blogId" ... />. The :blogId is a URL parameter.
2. **Linking (in BlogCard.js):** The "Read More" button is a <Link> component: <Link to={'/blog/12345'}>.
3. **Displaying (in BlogPostPage.js):** This component uses the useParams() hook from react-router-dom to extract blogId from the URL. It then filters the main allBlogs array to find the specific blog object and render its content.

4.2 Word .docx Upload and Parsing

This is the most complex feature of the application.

1. **The Input:** The BlogModal.js component has a hidden <input type="file"> that is triggered by a styled button.
2. **File Reading:** When a file is selected, an onChange handler is triggered. We use the browser's FileReader API to read the .docx file as an ArrayBuffer.
3. **Parsing with mammoth.js:** This ArrayBuffer is passed to mammoth.convertToHtml(). This function parses the document *in the browser*.
4. **Image Handling:** We critically use convertToHtml (not extractRawText) because it identifies images within the document. It converts these images into **Base64-encoded strings** and embeds them directly into the HTML within an <img> tag (e.g., <img src="...">).
5. **State Update:** The resulting HTML string (complete with text and embedded images) is stored in the content state. The file name is used to pre-fill the title state.

4.3 Rendering Unsafe HTML

Because the blog content is now a full HTML string, it cannot be rendered normally (React would just display the text Hello).

- **The Solution:** We use dangerouslySetInnerHTML={{ __html: blog.content }} in BlogPostPage.js.

- **Security:** This is normally a major security risk (Cross-Site Scripting, XSS). However, in this specific case, it is considered safe because the HTML is being generated *by the user, from their own file, on their own machine*. No one else can inject malicious code.

---

5. Challenges and Critical Limitations

The project's primary limitation is the technology chosen for its "database": **localStorage**.

- **The Challenge:** localStorage has a very small, browser-enforced quota (typically **5-10 MB** per website).
- **The Problem:** Base64-encoded images are extremely large, often 33% larger than the original image file.
- **The Consequence:** Uploading a *single* Word document with a few high-resolution images can generate enough Base64 text to **instantly fill the entire 5 MB quota**.
- **The Result:** The browser will throw a "QuotaExceededError," and the application will fail to save any more data. This limit **cannot** be increased by the application's code.

This limitation means the app, in its current state, is only suitable for text-based blogs or blogs with very small, few images.

---

6. Conclusion and Future Work

6.1 Conclusion

"Blogify" successfully achieves its goal of being a feature-rich, client-side blogging application. It demonstrates the power of modern React and JavaScript APIs to handle complex tasks like file parsing and dynamic routing without a server. The Word-to-HTML feature is a powerful proof-of-concept.

However, the project also serves as a critical case study on the limits of frontend-only architectures, especially regarding data storage.

6.2 Future Work

To evolve this project from a "demo" into a scalable, production-ready application, the localStorage limitation must be overcome. This requires a move to a full-stack architecture.

1. **Implement a Backend Server (Node.js/Express):** This server would handle file uploads.
2. **Add a Real Database (MongoDB/PostgreSQL):** The server would store blog post text/HTML in a database, which has no practical size limit.
3. **Implement File Storage (e.g., AWS S3 or a server folder):** This is the most important step. Instead of Base64, the server would extract images, save them as separate .jpg/.png files, and store them in the cloud. The database would then only store the *links* to these images (e.g., <img src="https.../my-image.png">). This is far more efficient.
4. **Add User Authentication:** A backend would allow for secure user login, so multiple users could store their own blogs.