TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

DBMS PROJECT REPORT ON POSTIFY BLOGGING PLATFORM

SUBMITTED BY	SUBMITTED TO
Abhishek Pachhain (078BEI001) **Bikash Pokhrel** (078BEI010) **Biplob Giri** (078BEI011)	Department of Electronics and Computer Engineering
Submission Date: 2081-04-25	**Signature:** ____________________

ABSTRACT

The project report presents an implementation of Postify, a blogging platform that emphasizes effective database management using MySQL and a backend built with Node.js/Express.js. The project explores the creation of a relational database structure to manage users, posts, comments, and categories efficiently. The objective is to implement a system that supports essential blogging functionalities, ensuring data integrity, scalability, and optimized query performance. The report discusses the design and implementation of the database schema, focusing on the use of SQL for data manipulation, the integration of the database with the backend, and the handling of user interactions. Additionally, it analyzes the system's performance, including query efficiency and scalability, and presents potential improvements and future enhancements to the platform.

ACKNOWLEDGEMENT

First of all, We would like to express our gratitude to our teacher, Anku Jaiswal, for her constant support and guidance throughout the semester. The lectures, assignments and guidelines have been of great help for us to successfully complete our project. We are also grateful to our lab teachers, Prakash Chandra Prasad and Sijal Baral, for their assistance and feedback throughout the practical sessions.

We would also like to thank Institute of Engineering, Pulchowk campus and Department of Electronics and Computer Engineering for providing us an opportunity to work on this project. Our friends and seniors have also been incredibly helpful in figuring out this whole ”teamwork for a project” scenario by sharing their knowledge and critique, this helped in shaping the outline and rectifying the elements of our project. We feel immensely thankful to be surrounded by such supportive, kind and obliging people. Our family has also played a huge role by providing us the support we needed-both emotional and mental-without which finishing the project would have been impossible. We would like to sincerely thank everyone who has directly or indirectly contributed to the project.

TABLE OF CONTENTS

ABSTRACT 1
ACKNOWLEDGEMENT 2
TABLE OF CONTENTS 3
INTRODUCTION 4
  Project Overview 4
  Objectives 4
  Structured Query Language(SQL) 4
  MySQL DBMS 4
METHODOLOGY 6
  System Architecture 6
  Frontend 6
  Backend 7
DATABASE DESIGN 8
  Overview 8
  Entity Relationship (ER) diagram 8
  Database Connection 8
  Models Overview 10
  User Model 10
  Post Model 14
  Category Model 16
  Comment Model 17
  PostCategory Model 19
OUTPUT 20
DISCUSSION 23
CONCLUSION 23

INTRODUCTION

Database Management Systems (DBMS) are essential tools for managing and organizing data in a structured manner. They provide a framework for storing, retrieving, and manipulating data efficiently while ensuring data integrity and security. A DBMS al lows users to interact with databases through a variety of operations, including querying, updating, and managing transactions. With features such as data redundancy control, access management, and backup facilities, DBMSs are fundamental to applications ranging from simple data storage to complex transaction processing systems

Project Overview

Postify is a dynamic blogging platform designed to offer an intuitive solution for content management. The project focuses on developing a robust database schema to handle various aspects of a blogging website, including posts, users, comments, and categories. The system leverages a well-structured database design to ensure efficient data retrieval, manipulation, and management. By integrating advanced features such as user authentication, comment management, and category organization, Postify aims to provide a seamless experience for both content creators and readers.

Objectives

The primary objective of the Postify project is to create a comprehensive blogging plat form with an emphasis on database management. The project aims to develop a detailed schema that supports core functionalities such as creating, reading, updating, and deleting posts, managing user interactions, and organizing content into categories. In addition to designing a functional database schema, the project will also focus on implementing SQL queries to handle complex operations, such as retrieving posts with associated user details and counting comments. By providing a user-friendly interface and efficient data handling capabilities, Postify seeks to enhance the overall user experience and ensure reliable content management. This project serves as a practical application of database design principles and SQL querying techniques, demonstrating their real-world relevance and effectiveness.

Structured Query Language(SQL)

Structured Query Language (SQL) is a powerful and standardized language used for managing and manipulating relational databases. It enables users to perform various operations such as querying data, updating records, inserting new data, and deleting data. SQL provides a systematic way to interact with databases through commands such as SELECT, INSERT, UPDATE, and DELETE, making it an essential tool for database administrators and developers. Its declarative nature allows users to specify what data to retrieve or manipulate without needing to describe how to achieve it, thus streamlining complex data management tasks.

MySQL DBMS

MySQL is a widely adopted open-source relational database management system (RDBMS) known for its robustness, performance, and ease of use. It uses SQL as its query language to facilitate efficient data management and retrieval. MySQL is popular for its reliability and scalability, making it suitable for a range of applications from small web projects to large-scale enterprise systems. It supports various features such as data indexing, transactions, and data security, and is often chosen for its compatibility with numerous platforms and integration with various programming languages.

METHODOLOGY

System Architecture

The *Postify* application is designed with a modern web architecture leveraging ReactJS for the frontend and Node.js with Express.js for the backend, with MySQL serving as the relational database management system. The application follows the Model-View-Controller (MVC) architecture, ensuring a clear separation of concerns and maintainability.

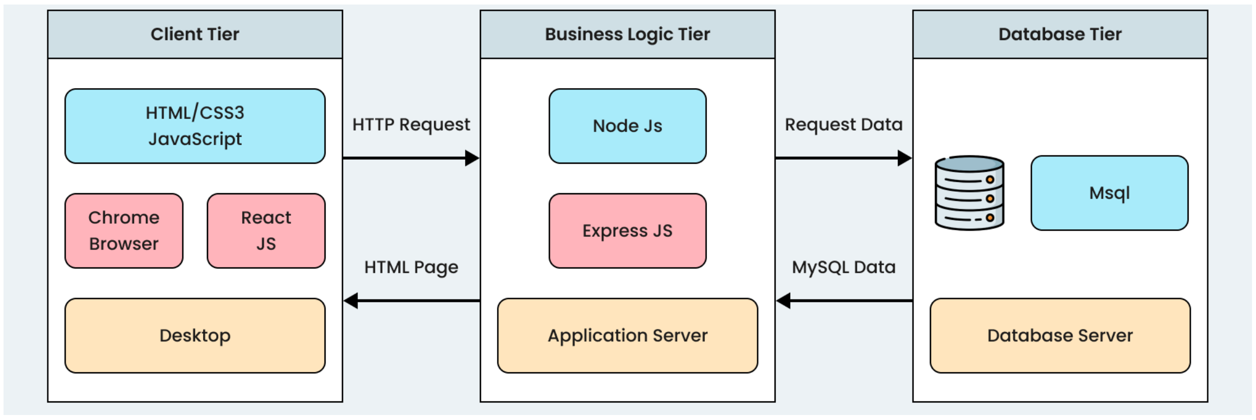


Fig: High Level System Architecture

Frontend

The frontend is built using HTML, CSS and ReactJS, a popular JavaScript library for building user interfaces. ReactJS facilitates the creation of a dynamic and responsive user experience. The application is structured with several routes handled by React Router, including public and private routes. Public routes are accessible to all users and include views such as the landing page, home, and blog posts. Private routes are protected and require authentication, granting access to user-specific features such as profile management and blog creation/editing.

The frontend is structured with the following routes:

1. **/:** The root route which renders the LandingPage component. It serves as the entry point of the application.

2. **/home**: Displays the Home component, which is the main page where users can view a list of blogs.

3. **/signup**: Shows the SignUp component for user registration.

4. **/signin**: Displays the SignIn component for user login.

5. **/blog/:blogId**: Renders the BlogPost component, which shows a specific blog post based on the blog ID in the URL.

6. **/profile**: Protected route that renders the Profile component, allowing users to view and edit their profile information.

7. **/create-blog**: Protected route for creating a new blog post, rendered by the AddBlog component.

8. **/personal-page**: Protected route showing the PersonalPage component, where users can manage their personal blogs.

9. **/edit-blog/:blogId**: Protected route that allows users to edit an existing blog post based on the blog ID, using the EditBlog component.

Backend

The backend of the Postify application is developed using Node.js and Express.js, following the Model-View-Controller (MVC) architecture. The outline of the backend is:

- **Model**: Manages the data and database interactions. Models are responsible for handling CRUD operations and business logic related to data entities such as posts, comments, and users. The MySQL database is used for data storage, with SQL queries implemented to perform operations on the database.

- **Controller**: Contains the logic to handle requests from the frontend, process them using the models, and return appropriate responses. Controllers manage the application's workflow and handle user input, interfacing between the models and the views (ReactJS components).

- **Routes**: Define the API endpoints that the frontend communicates with. Routes are managed using Express.js to handle requests and route them to the appropriate controllers.

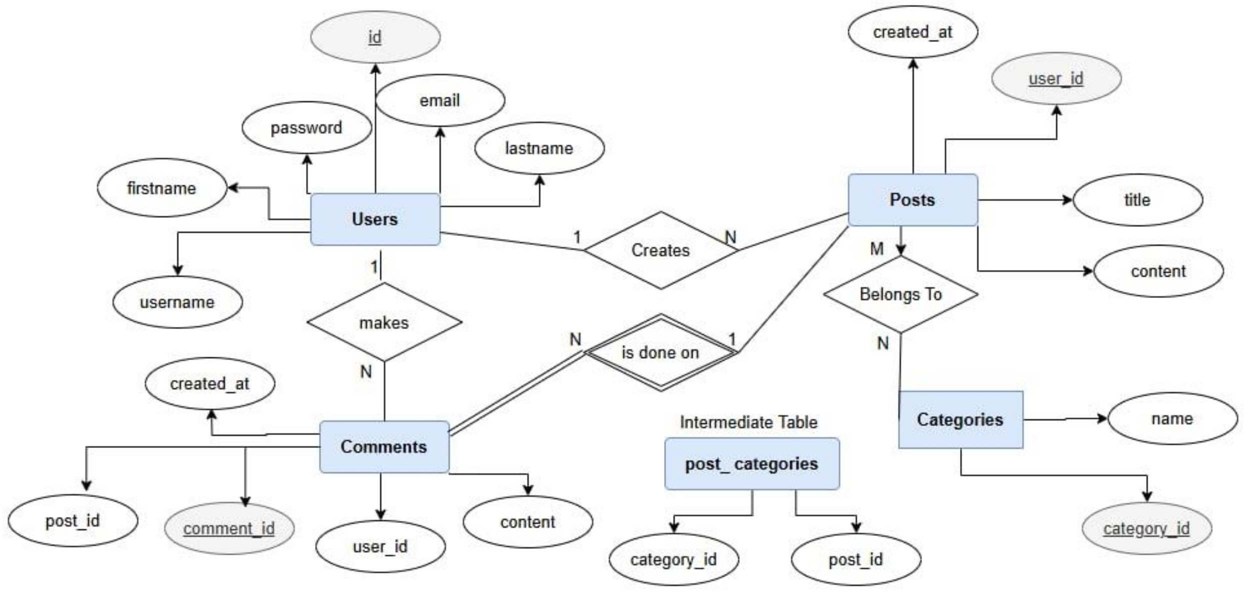
- **Database**: MySQL is used as the relational database management system (RDBMS). It handles data storage and retrieval, with SQL being used to perform operations such as querying, updating, and managing data.

DATABASE DESIGN

Overview

Since the course focuses on the database aspect, the explanation will be oriented toward the database structure and interactions. The project uses MySQL as the backend database, with the database connection managed through Node.js and the ORM-like interactions facilitated by custom model files for each table. The entire database interaction is encapsulated within these models, which handle everything from table creation to data manipulation.

Entity Relationship (ER) diagram



Database Connection

The connection to the MySQL database is handled in the Database.js file. The connection is created using mysql2/promise, which provides a promise-based API for MySQL interactions. Here's a high-level overview of the connection process:

1. **Configuration:** Database credentials like host, user, password, and database name are loaded from environment variables.

2. **Connection Pool:** A connection pool is established using mysql.createPool(), which allows for efficient management of multiple simultaneous connections.

3. **Query Execution:** The query() method allows execution of SQL queries with parameters, ensuring secure and efficient database operations.

CODE: Database.js

```
import mysql from "mysql2/promise";
import dotenv from "dotenv";

dotenv.config();

class Database {
  // Connection details are initialized here.
  constructor({ host, user, password, database }) {
    this.host = host;
    this.user = user;
    this.password = password;
    this.database = database;
  }

  // Establishing a connection pool to handle database queries.
  async connect() {
    try {
      this.connection = mysql.createPool({
        host: this.host,
        user: this.user,
        password: this.password,
        database: this.database,
        waitForConnections: true,
        connectionLimit: 10,
        queueLimit: 0,
      });

      console.log("Successfully connected to the database.");
    } catch (error) {
      console.error("Error connecting to the database:", error);
      throw error;
    }
  }

  // Method to execute SQL queries.
  async query(sql, params) {
    try {
      if (!this.connection) {
        throw new Error("Database connection not established.");
      }
      const   [results]   =   await   this.connection.execute(sql,
params);
      return results;
    } catch (error) {
      console.log(error);
    }
  }

  // Closing the database connection.
  async close() {
    await this.connection.end();
  }
}

export const db = new Database(dbConfig); // Creating a new
instance of Database.
```

Models Overview

Each table in the database is represented as a model file in the project. The models encapsulate the table structure (schema) and methods to interact with the database (CRUD operations). The key models include User, Post, Comment, Category, and PostCategory. The PostCategory model handles the many-to-many relationship between posts and categories.

User Model

The User.model.js file represents the users table in the database. This model handles user-related operations, such as creating a new user, updating user details, finding users by various criteria, and more.

Schema:

Column Name	**Data Type**	**Constraints**	**Description**
id	CHAR(36)	PRIMARY KEY, NOT NULL	Unique identifier for each user (UUID).
firstname	VARCHAR(50)	NOT NULL	First name of the user.
lastname	VARCHAR(50)	NOT NULL	Last name of the user.
username	VARCHAR(50)	NOT NULL, UNIQUE	Unique username for the user.
email	VARCHAR(100)	NOT NULL, UNIQUE	Unique email address of the user.

password	VARCHAR(255)	NOT NULL	Hashed password for the user.

SQL Queries and Operations

1. **Table Creation**
   The initialize() method creates the users table if it doesn't already exist.

```
const sql = `CREATE TABLE IF NOT EXISTS users (
    id CHAR(36) PRIMARY KEY NOT NULL,
    firstname VARCHAR(50) NOT NULL,
    lastname VARCHAR(50) NOT NULL,
    username VARCHAR(50) NOT NULL UNIQUE,
    email VARCHAR(100) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL
);`;

await db.query(sql);
```

2. **Insert or Update User:**
   The save() method inserts a new user or updates an existing user if the id is already present.

```
if (this.id) {
  await db.connection.query(
    "UPDATE users SET firstname = ?, lastname = ?, username
= ?, email = ?, password = ? WHERE id = ?",
    [this.firstname,     this.lastname,     this.username,
this.email, this.password, this.id]
  );
} else {
  await db.connection.query(
    "INSERT  INTO  users  (firstname,  lastname,  username,
email, password) VALUES (?, ?, ?, ?, ?)",
    [this.firstname,     this.lastname,     this.username,
this.email, this.password]
  );
}
```

3. **Find User by Id**
   The findById() method retrieves a user's details based on the provided id.

```
const [rows] = await db.connection.query(
  "SELECT * FROM users WHERE id = ?",
  [id]
);
```

4. **Find User by Email and Password**
   The findByEmailAndPassword() method is used to authenticate a user by verifying their email and hashed password. It calls a stored procedure named SelectUserWithHashedPassword, which takes the provided email and plain-text password as inputs, hashes the password using SHA-256, and then checks for a match in the users table.

   ```
   const [rows] = await db.connection.query(
     "CALL SelectUserWithHashedPassword(?, ?)",
     [email, password]
   );
   ```

Stored Procedures

SelectUserWithHashedPassword

This stored procedure is used to authenticate a user by verifying their email and password. The process involves hashing the plain-text password using the SHA-256 algorithm and then comparing the hashed password with the stored password in the database.

CODE:

```
DELIMITER $$

CREATE PROCEDURE SelectUserWithHashedPassword (
    IN p_email VARCHAR(100),
    IN p_password VARCHAR(255)
)
BEGIN
    DECLARE hashed_password VARCHAR(255);
    SET hashed_password = SHA2(p_password, 256); -- Assuming SHA-
256 is used for hashing

    SELECT * FROM users WHERE email = p_email AND password =
hashed_password;
END $$

DELIMITER ;
```

Triggers

before_user_update

This trigger automatically hashes the user's password before an update operation if the password has been changed. It ensures that passwords are always stored in a hashed format.

- **Event:** This trigger is executed before an UPDATE operation on the users table.
- **Condition:** It checks if the NEW.password (the updated password) is different from the OLD.password (the current password in the database).
- **Action:** If the password has changed, the trigger hashes the new password using SHA-256 before saving it to the database.

CODE:

```
DELIMITER $$

CREATE TRIGGER before_user_update
BEFORE UPDATE ON users
FOR EACH ROW
BEGIN
  IF NEW.password != OLD.password THEN
    SET NEW.password = SHA2(NEW.password, 256);
  END IF;
END $$

DELIMITER ;
```

before_user_insert

This trigger automatically hashes the user's password and generates a UUID for the user ID before a new record is inserted into the users table.

- **Event:** This trigger is executed before an INSERT operation on the users table.
- **Actions:**
    1. Hashes the NEW.password using SHA-256 to ensure secure storage.
    2. If NEW.id is NULL, the trigger generates a unique identifier (UUID) for the new user.

CODE:

```
DELIMITER $$

CREATE TRIGGER before_user_insert
BEFORE INSERT ON users
FOR EACH ROW
```

```
BEGIN
  SET NEW.password = SHA2(NEW.password, 256);
  IF NEW.id IS NULL THEN
    SET NEW.id = UUID();
  END IF;
END $$

DELIMITER ;
```

Post Model

Schema:

Column Name	Data Type	Constraints	Description
post_id	INT	AUTO_INCREMENT, PRIMARY KEY, NOT NULL	Unique identifier for each post.
user_id	CHAR(36)	NOT NULL, FOREIGN KEY	References the user who created the post.
title	VARCHAR(255)	NOT NULL	Title of the post.
content	TEXT	NOT NULL	Content of the post.
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp of when the post was created.

SQL Queries and Operations

1. **Table Creation**
   This query creates the posts table, defining its structure and constraints.

```
CREATE TABLE IF NOT EXISTS Posts (
    post_id INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
    user_id CHAR(36) NOT NULL,
    title VARCHAR(255) NOT NULL,
    content TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id),
    FOREIGN        KEY        (category_id)        REFERENCES
Categories(category_id)
);
```

2. **Insert or Update Post**
   The following SQL queries are used to either insert a new post or update an existing one based on whether the post_id is present.

   ```
   INSERT INTO Posts (user_id, title, content) VALUES (?, ?,
   ?);
   UPDATE Posts SET user_id = ?, title = ?, content = ? WHERE
   post_id = ?;
   ```

3. **Finding a Post by ID**
   This query retrieves a post by its post_id.

   ```
   SELECT * FROM Posts WHERE post_id = ?;
   ```

4. **Find Post by ID with Details:**
   This query retrieves a post along with associated user information, comments, and categories.

   ```
   SELECT p.*,
          u.firstname, u.lastname, u.username, u.email,
          c.comment_id, c.user_id AS comment_user_id, c.content
   AS comment_content, c.created_at AS comment_created_at,
          pc.category_id, cat.name AS category_name,
          (SELECT COUNT(*) FROM comments WHERE comments.post_id
   = p.post_id) AS comment_count
   FROM posts p
   JOIN users u ON p.user_id = u.id
   LEFT JOIN comments c ON p.post_id = c.post_id
   LEFT JOIN post_categories pc ON p.post_id = pc.post_id
   LEFT JOIN categories cat ON pc.category_id = cat.category_id
   WHERE p.post_id = ?;
   ```

5. **Find Posts with User ID:**
   This query retrieves all posts created by a specific user, including user details and the count of comments on each post.

   ```
   SELECT
       p.*,
       u.firstname, u.lastname, u.username, u.email,
   ```

```
    c.comment_id, c.user_id AS comment_user_id, c.content AS
comment_content, c.created_at AS comment_created_at,
    (SELECT COUNT(*) FROM comments WHERE comments.post_id =
p.post_id) AS comment_count
FROM posts p
JOIN users u ON p.user_id = u.id
LEFT JOIN comments c ON p.post_id = c.post_id
WHERE u.id = ?;
```

6. **Delete Post by ID:**
   This query deletes a post from the database using its post_id.

   ```
   DELETE FROM Posts WHERE post_id = ?
   ```

7. **Count the number of posts**
   This query count the number of posts in post table.

   ```
   SELECT COUNT(*) AS count FROM posts
   ```

8. **Add Categories to a Post:**
   This query inserts categories into the post_categories table, associating them with a specific post.

   ```
   INSERT INTO post_categories (post_id, category_id)
   VALUES (?, ?), (?, ?), ...;
   ```

9. **Update Categories for a Post**
   This query deletes all existing category associations for a post and then adds new categories.

   ```
   -- Delete existing categories for a post
   DELETE FROM post_categories WHERE post_id = ?;

   -- Add new categories to a post
   INSERT INTO post_categories (post_id, category_id)
   VALUES (?, ?), (?, ?), ...;
   ```

Category Model

Schema:

Column Name	**Data Type**	**Constraints**	**Description**

category_id	INT	AUTO_INCREMENT, PRIMARY KEY, NOT NULL	Unique identifier for each category.
name	VARCHAR(255)	NOT NULL	Name of the category.

SQL Queries and Operations

1. **Table Creation**
   This query creates the categories table, defining its structure and constraints.

```
CREATE TABLE IF NOT EXISTS categories (
    category_id INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
    name VARCHAR(255) NOT NULL
);
```

2. **Find a Category by ID with Posts**
   This query retrieves a category by its category_id along with the associated posts, user details, and comment count.

```
SELECT
    c.category_id, c.name,
    p.post_id, p.user_id, p.title, p.content, p.created_at,
    u.firstname, u.lastname, u.username, u.email,
    (SELECT COUNT(*) FROM comments WHERE comments.post_id =
p.post_id) AS comment_count
FROM categories c
LEFT   JOIN   post_categories   pc   ON   c.category_id   =
pc.category_id
LEFT JOIN posts p ON pc.post_id = p.post_id
LEFT JOIN users u ON p.user_id = u.id
WHERE c.category_id = ?;
```

3. **Find all categories**
   This query retrieves all categories from the categories table.

```
SELECT * FROM categories;
```

Comment Model

Schema:

Column Name	**Data Type**	**Constraints**	**Description**
comment_id	INT	AUTO_INCREMENT, PRIMARY KEY, NOT NULL	Unique identifier for each comment.
post_id	INT	NOT NULL, FOREIGN KEY	References the post to which the comment belongs.
user_id	CHAR(36)	NOT NULL, FOREIGN KEY	References the user who made the comment.
content	TEXT	NOT NULL	Content of the comment.
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp of when the comment was created.

SQL Queries and Operations

1. **Table Creation**
   This query creates the comments table, defining its structure and constraints.

```
CREATE TABLE IF NOT EXISTS comments (
    comment_id INT AUTO_INCREMENT PRIMARY KEY,
    post_id INT NOT NULL,
    user_id CHAR(36) NOT NULL,
    content TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (post_id) REFERENCES Posts(post_id),
    FOREIGN KEY (user_id) REFERENCES Users(id)
);
```

2. **Find a comment by id**
   This query retrieves a comment by its comment_id.

```
SELECT * FROM comments WHERE comment_id = ?;
```

3. **Find comments by Post Id**
   This query retrieves all comments for a specific post along with user information and the total count of comments.

```
SELECT c.*,
       u.firstname, u.lastname, u.username, u.email,
       COUNT(*) OVER() AS total_comments
FROM comments c
JOIN users u ON c.user_id = u.id
WHERE c.post_id = ?
ORDER BY c.created_at DESC;
```

4. **Save or Update Comment**

```
INSERT INTO comments (user_id, post_id, content)
VALUES (?, ?, ?);

UPDATE comments SET content = ?
WHERE comment_id = ?;
```

PostCategory Model

The PostCategory model represents a junction table used to manage the many-to-many relationship between Posts and Categories. Each post can be associated with multiple categories, and each category can include multiple posts. This model facilitates this relationship by using a linking table called post_categories.

Schema:

Column Name	**Data Type**	**Constraints**	**Description**
post_id	INT	PRIMARY KEY, FOREIGN KEY	References the post in the posts table.
category_id	INT	PRIMARY KEY, FOREIGN KEY	References the category in the categories table.

SQL Queries and Operations:

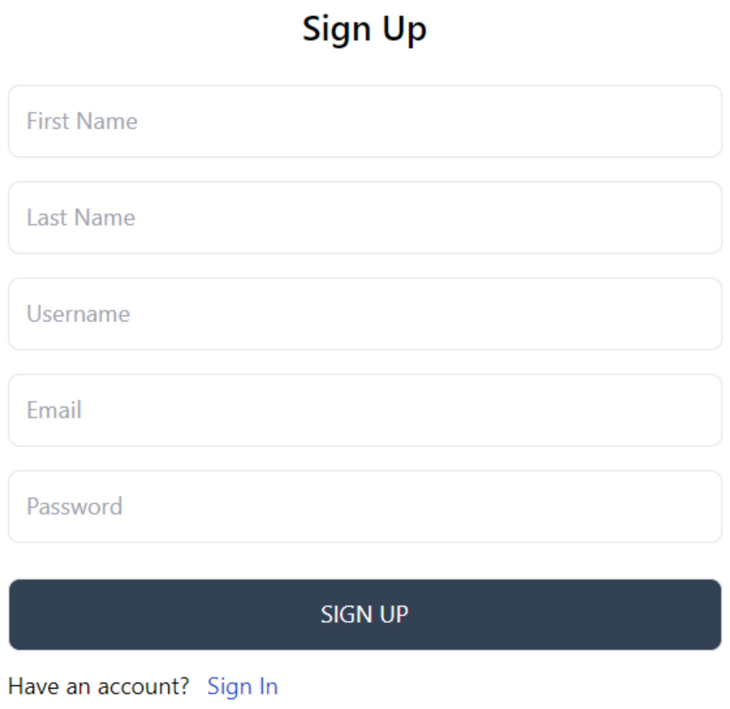
1. **Table Creation**
   This query creates the post_categories table with a composite primary key consisting of post_id and category_id. It also sets up foreign key constraints to ensure that each

post_id and category_id in the table corresponds to existing records in the posts and categories tables, respectively. The ON DELETE CASCADE option ensures that if a post or category is deleted, the associated records in post_categories are automatically removed.

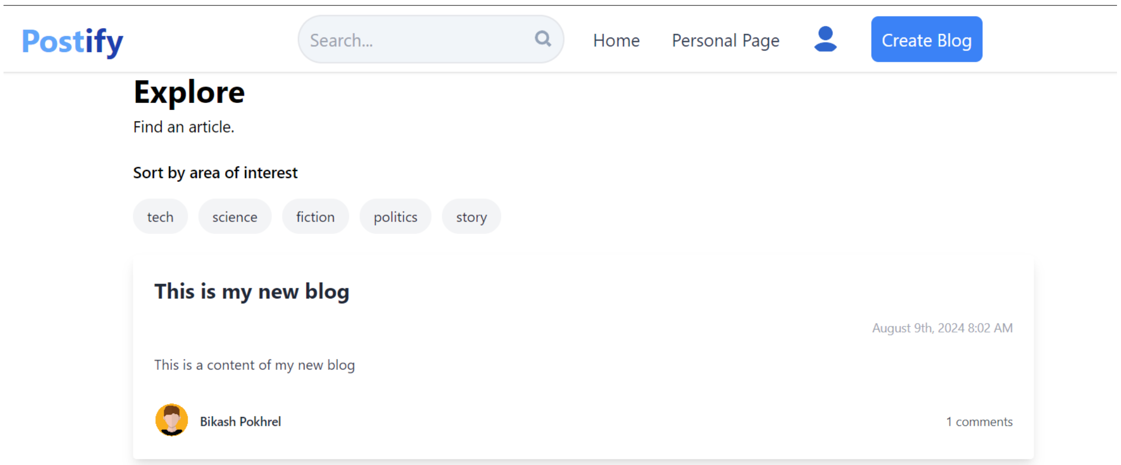
```
CREATE TABLE IF NOT EXISTS post_categories (
    post_id INT,
    category_id INT,
    PRIMARY KEY (post_id, category_id),
    FOREIGN KEY (post_id) REFERENCES posts(post_id) ON
DELETE CASCADE,
    FOREIGN KEY (category_id) REFERENCES
categories(category_id) ON DELETE CASCADE
);
```

OUTPUT

Signup Page



Home Page



Explore

Find an article.

Sort by area of interest

tech science fiction politics story

This is my new blog

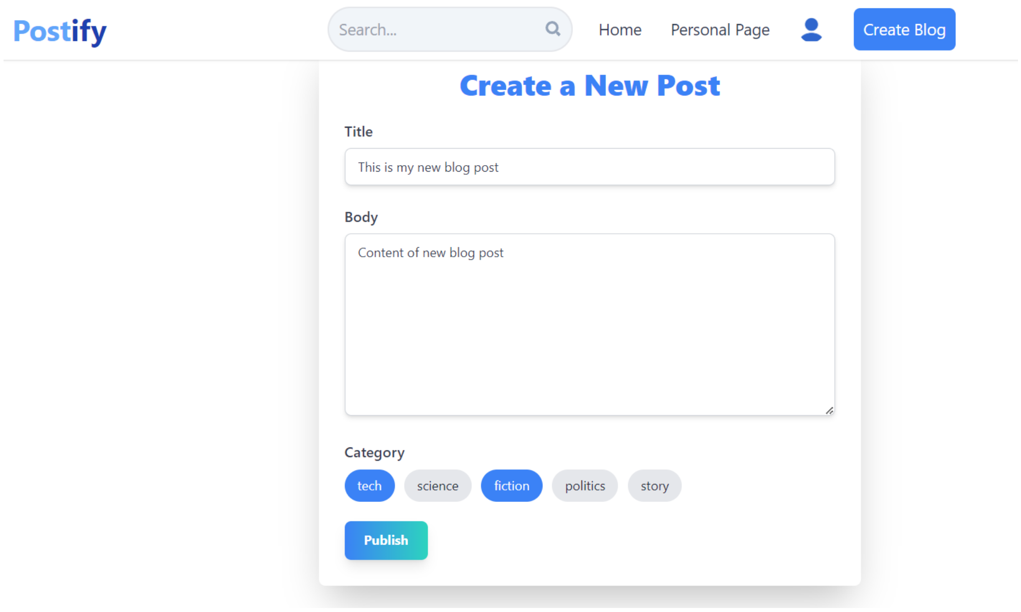
August 9th, 2024 8:02 AM

This is a content of my new blog

Bikash Pokhrel

1 comments

Create Blog Page



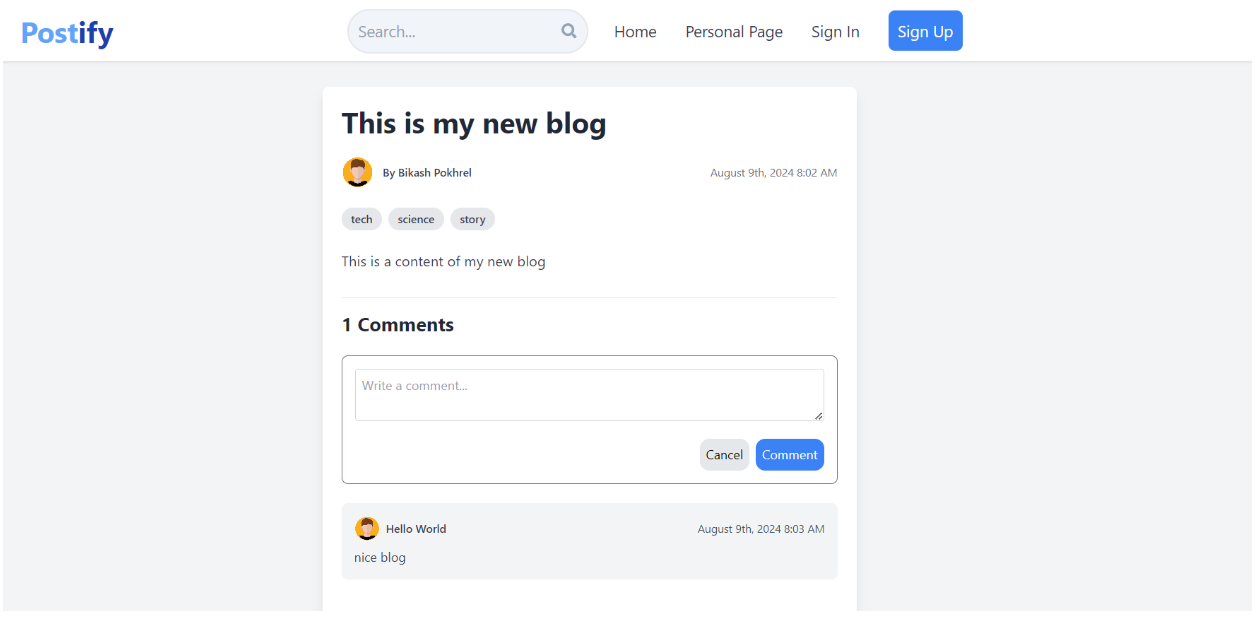
Create a New Post

Title: This is my new blog post

Body: Content of new blog post

Category: tech science fiction politics story

Post Page



This is my new blog

By Bikash Pokhrel

August 9th, 2024 8:02 AM

tech science story

This is a content of my new blog

1 Comments

Hello World

August 9th, 2024 8:03 AM

nice blog

DISCUSSION

The development of Postify has provided invaluable insights into the intricacies of database management systems and their integration with modern web technologies. Throughout this project, we have successfully implemented a full-stack application that leverages ReactJS for the frontend, Node.js/Express.js for the backend, and MySQL as the database. The experience gained from this project has reinforced the importance of a well-structured database design and the practical application of SQL for managing data efficiently.

One of the key learnings from this project is the significance of adhering to the principles of the MVC (Model-View-Controller) architecture. By separating concerns between data handling, application logic, and user interface, the project has demonstrated the benefits of modularity and maintainability in software development. Furthermore, working with MySQL has deepened our understanding of relational database concepts, including schema design, normalization, and the execution of complex SQL queries.

While the current implementation of Postify meets the project's objectives, there are several areas for further enhancement:

1. **Scalability**: As the application grows, optimizing database queries and introducing indexing strategies will be crucial for maintaining performance. Implementing caching mechanisms, such as Redis, can also help in reducing database load and improving response times.

2. **Enhanced User Experience**: The user interface can be further refined by incorporating more dynamic and interactive elements, such as real-time notifications for new posts or comments using WebSockets.

3. **Security**: Although basic security measures are in place, such as authentication and input validation, the project can benefit from implementing more robust security practices, including encryption of sensitive data, role-based access control, and protection against common web vulnerabilities like SQL injection and cross-site scripting (XSS).

CONCLUSION

Hence, the Postify project deepened our understanding of database management by focusing on designing and implementing a relational database using MySQL. We learned how to structure data effectively, optimize SQL queries, and maintain data integrity, which are crucial for building scalable web applications. The project also highlighted the importance of efficient database interactions within a Node.js/Express.js backend, reinforcing best practices in database design and management.