数据库实验课

选做项目

作业名称：基于 Flask+Vue 的现代化论坛数据库系统

组　长：_____

组　员：_____

组　员：_____

学　院：_____

班　级：_____

摘要

本项目设计并实现了一个基于前后端分离架构的现代化在线论坛系统。系统以 PostgreSQL 为数据库管理系统，后端采用 Python Flask 框架构建 RESTful API，利用 Flask-SQLAlchemy 进行数据持久化，并通过 Flask-Migrate 实现数据库版本控制；前端则基于 Vue.js 3 构建动态、响应式的单页应用（SPA），通过 Axios 与后端进行异步数据交互。项目完整实现了用户认证（JWT）、帖子发布与管理（CRUD）、实时评论等核心功能，并通过前后端双重校验确保了操作的权限安全。在开发过程中，完整经历了从需求分析、E-R 图设计、关系模型转换，到数据库实施、全栈编码，最终成功将应用分别部署至 Render 和 Vercel 云平台，实现了 CI/CD 自动化流程。

该项目不仅全面地实践了数据库课程设计的核心要求，更深入探索了现代 Web 应用开发的全链路工程实践，有效提升了在复杂系统设计、开发与部署方面的综合能力。

关键词

数据库系统, PostgreSQL, 前后端分离, Flask, Vue.js, RESTful API, ORM, 数据库迁移

目录

第一章：引言

1.1 选题背景与意义

在数字化信息时代，在线社区与论坛作为网络社会的重要组成部分，扮演着不可或-缺的角色。它们不仅是个人用户分享观点、交流思想、构建兴趣社群的关键平台，也是企业与组织进行用户沟通、知识沉淀和品牌建设的重要阵地。从技术问答社区 Stack Overflow 到综合性讨论平台 Reddit，成功的在线论坛极大地促进了信息的自由流动和知识的集体创造，其社会与经济价值日益凸显。

然而，许多传统的论坛系统在技术架构上仍停留在较为陈旧的整体式（Monolithic）模式。此类系统通常将前端界面渲染与后端业务逻辑紧密耦合，导致了诸多不足：首先，用户体验受限，页面的每次局部更新都需要整个页面的刷新，交互流畅度差，难以满足现代用户对“即时响应”的高要求；其次，系统可扩展性与可维护性差，前后端代码混杂，使得独立开发、测试和部署变得困难，技术栈升级缓慢，难以快速迭代以适应新的业务需求或引入新的终端（如移动 App）。

鉴于此，采用现代化的前后端分离架构来构建论坛系统具有重要的实践意义。该架构将系统解耦为独立的前端应用和后端服务，通过标准化的 API (应用程序编程接口) 进行通信。这种模式能够带来显著优势：一是极致的用户体验，前端可以构建为单页应用（SPA），实现页面内容的动态、无刷新更新，响应速度更快，交互更流畅；二是强大的可扩展性，后端 API 作为纯粹的数据服务，可以被 Web 端、移动端、桌面端等多个客户端复用，实现了“一次开发，多端使用”；三是高效的开发流程，前后端团队可以并行开发、独立部署，大大提升了开发效率和系统的灵活性。

因此，本项目旨在基于上述先进理念，设计并实现一个功能完善、技术先进的现代化论坛数据库应用系统。通过本次课程设计，我们将深入探索从数据库底层设计到全栈应用开发的完整流程，构建一个既满足核心业务需求，又具备良好用户体验和技术前瞻性的实战项目。

1.2 项目目标

为实现一个高质量的现代化论坛系统，本项目设定了以下四个核心目标，覆盖了从底层数据结构到顶层用户交互的全过程：

1. **设计健壮、规范的数据库模型：** 这是整个系统的基石。我们将依据需求分析，设计出符合数据库范式、关系清晰、能够高效支撑论坛核心业务（用户、帖子、评论）的数据库逻辑模型和物理模型，并确保其具有良好的数据完整性和可扩展性。
2. **实现一个基于 RESTful API 的后端服务：** 我们将利用 Flask 框架构建一个无状态（Stateless）的后端应用。该应用的核心职责是提供一套标准化的 RESTful API 接口，用于处理前端的数据请求，执行所有与数据库相关的业务逻辑操作（增删改查），并确保接口的安全性和权限控制。
3. **开发一个动态、响应式的前端用户界面：** 我们将采用主流的 Vue.js 框架构建一个单页应用（SPA）。该前端应用负责所有用户的视图渲染和交互逻辑，通过异步调用后端 API 来获取和提交数据，为用户提供快速、流畅、无需频繁刷新页面的现代化浏览体验。
4. **完成一个从数据库设计到全栈应用开发部署的完整闭环：** 本项目的最终目标是打通数据库课程设计的全链路。我们将完整经历需求分析、数据库设计（概念、逻辑、

物理）、数据库实施、后端 API 开发、前端 UI 开发，直至最终将应用部署到云平台的全过程，形成一个理论与实践紧密结合的完整项目成果。

1.3 技术栈概述

为达成项目目标，我们选用了一套在业界广泛应用且技术成熟的现代化技术栈，具体如下：

1、 数据库管理系统 (DBMS):

PostgreSQL: 一款功能强大、性能卓越的开源对象-关系型数据库系统。以其高并发性、数据一致性和对复杂查询的良好支持而著称，非常适合作为论坛这类读写密集型应用的数据存储后端。

2、 后端技术 (Backend):

a) **Python:** 一种语法简洁、生态丰富的通用编程语言。
b) **Flask:** 一个轻量级的 Python Web 框架，以其灵活性和最小化的核心而闻名，非常适合用于快速构建 RESTful API。
c) **Flask-SQLAlchemy:** Flask 的一个官方扩展，提供了强大的对象关系映射（ORM）能力，使我们能用 Python 类和对象来直观地操作数据库，而无需编写原生 SQL。
d) **Flask-Migrate:** 基于 Alembic 的数据库迁移工具，能够追踪数据模型的变化，并自动生成数据库结构变更的脚本，极大地简化了数据库的版本管理。

3、 前端技术 (Frontend):

a) **Vue.js 3:** 一套用于构建用户界面的渐进式 JavaScript 框架，以其响应式数据绑定和组件化系统而闻名，能够高效地构建复杂的单页应用。
b) **Vue Router:** Vue.js 的官方路由管理器，用于实现前端页面之间的无刷新跳转。
c) **Vuex:** Vue.js 的官方状态管理模式和库，用于集中式存储和管理应用中所有组件共享的状态。
d) **Axios:** 一个基于 Promise 的 HTTP 客户端，用于在前端应用中发起对后端 API 的异步请求。

4、 开发工具 (Development Tools):

a) **Visual Studio Code (VS Code):** 一款功能强大的、跨平台的源代码编辑器。
b) **pgAdmin 4:** PostgreSQL 的官方图形化管理工具，用于数据库的设计、查询和维护。
c) **Git:** 一个分布式版本控制系统，用于项目的代码管理和团队协作。

第二章：系统需求分析

2.1 功能性需求分析

本论坛系统旨在为用户提供一个基础但完整的话题讨论平台。其核心功能性需求可划分为以下三个主要模块：

1. **用户模块 (User Module):** 这是系统的基础权限管理模块，保障了社区内容的归属和安全。

   用户注册： 新用户可以通过提供唯一的用户名、电子邮箱和密码来创建账户。

   用户登录：已注册用户可以通过用户名和密码进行登录，以获取发布和评论的权限。

   用户登出： 登录用户可以随时退出登录状态。

   身份认证： 系统采用基于 JSON Web Tokens (JWT) 的无状态认证机制。用户登录成功后，后端会签发一个有时效性的 Token，前端在后续的每次请求中都需携带此 Token，以证明其身份和操作权限。

2. **帖子模块 (Post Module):** 这是论坛的核心内容模块，承载了话题的发起和展示。

   创建帖子： 登录用户可以发布包含标题和正文内容的新帖子。

   删除帖子： 帖子的作者拥有删除自己所发布帖子的权限。

   帖子列表展示： 任何用户（包括未登录的访客）都可以在论坛首页看到所有帖子的列表，该列表应按发布时间的先后倒序排列，最新的帖子显示在最前面。

   编辑帖子： 作为系统的未来扩展方向，预留了帖子作者可以编辑自己帖子的功能接口。

3. **评论模块 (Comment Module):** 这是论坛的互动模块，提供了用户间交流的渠道。

   发表评论： 登录用户可以在任意一个帖子的详情页下方发表自己的评论。

   删除评论： 评论的作者可以删除自己的评论。

   评论列表展示： 在帖子详情页，系统应展示该帖子的所有评论，并按发表时间的先后顺序排列。

2.2 数据需求分析

为满足上述功能需求，我们对系统所需存储的数据进行分析，识别出核心的数据实体、它们的属性以及它们之间的关系。

1. **实体识别 (Entity Identification):**

   系统中最核心的业务对象可以抽象为三个实体：

   用户 (User): 论坛的参与者，是发布帖子和评论的主体。

   帖子 (Post): 用户发起的话题，是论坛内容的核心载体。

   评论 (Comment): 用户对帖子的回应，是论坛互动的主要形式。

2. **实体属性定义 (Attribute Definition):**

   用户 (User) 实体属性:

   ID: 用户的唯一标识符，系统内部使用。

   用户名 (username): 用户登录和显示的名称，必须唯一。

   电子邮箱 (email): 用于注册验证或找回密码，必须唯一。

   密码哈希值 (password_hash): 存储经哈希算法加密后的用户密码，而非明文密码，以保障账户安全。

帖子 (Post) 实体属性:

ID: 帖子的唯一标识符。

标题 (title): 帖子的标题。

内容 (content): 帖子的正文。

发布时间 (timestamp): 帖子的创建时间，用于排序。

作者 ID (user_id): 关联发布该帖子的用户 ID。

评论 (Comment) 实体属性:

ID: 评论的唯一标识符。

内容 (content): 评论的正文。

发布时间 (timestamp): 评论的创建时间。

评论者 ID (user_id): 关联发表该评论的用户 ID。

所属帖子 ID (post_id): 关联该评论所属的帖子 ID。

3. **实体间关系分析 (Relationship Analysis):**

用户 (User) 与 帖子 (Post) 的关系: 一个用户可以发布多篇帖子，但一篇帖子只能由一个用户发布。这是典型的 **一对多 (1:N)** 关系。

用户 (User) 与 评论 (Comment) 的关系: 一个用户可以发表多条评论，但一条评论只能由一个用户发表。这也是 **一对多 (1:N)** 关系。

帖子 (Post) 与 评论 (Comment) 的关系: 一篇帖子可以有多条评论，但一条评论只能属于一篇帖子。这同样是 **一对多 (1:N)** 关系。

2.3 性能与安全性需求

1. **性能需求 (Performance Requirements):**

快速加载： 论坛首页的帖子列表是访问最频繁的页面，必须保证快速响应。数据库查询应进行优化，例如为帖子的发布时间字段建立索引。

实时交互： 用户发表评论后，评论列表应能“实时”更新，无需刷新整个页面。这要求前后端接口设计高效，前端能够进行局部数据刷新。

2. **安全性需求 (Security Requirements):**

数据安全： 用户的密码绝不能以明文形式存储在数据库中，必须经过高强度的哈希算法（如 Werkzeug Security 提供的 generate_password_hash）处理后存储。

访问控制： 系统的核心写操作（如发帖、评论、删除）必须受到保护。所有需要权限的 API 接口都必须通过 JWT 进行身份验证。同时，后端逻辑必须进行二次校验，确保例如只有帖子的作者本人才能删除该帖子，防止恶意越权操作。

第三章：数据库设计

数据库设计是构建稳定、高效应用系统的核心环节。本章将遵循规范化的数据库设计流程，从概念层面的 E-R 图，到逻辑层面的关系模型，再到物理层面的具体实现，详细阐述本论坛系统的数据库设计过程。

3.1 概念结构设计（E-R 图）

概念结构设计的目标是从用户需求中抽象出核心的实体（Entity）、属性（Attribute）和关系（Relationship），并用 E-R 图（Entity-Relationship Diagram）进行可视化表达。根据第二章的需求分析，我们识别出“用户（User）”、“帖子（Post）”和“评论（Comment）”三个核心实体。

它们之间的关系如下：

1. 一个“用户”可以发布多篇“帖子”（**发布**关系）。
2. 一个“用户”可以发表多条“评论”（**发表**关系）。
3. 一篇“帖子”可以拥有多条“评论”（**拥有**关系）。

这三个关系均为典型的一对多（1:N）关系。其 E-R 图如下所示：

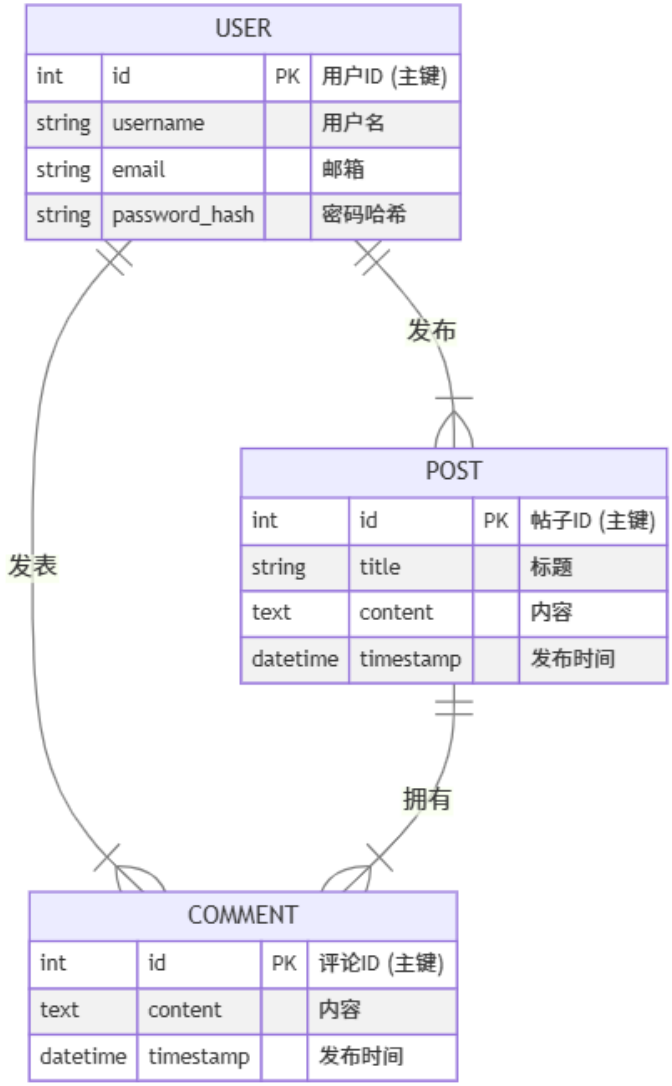

图 3-1 **论坛系统 E-R 图**

3.2 逻辑结构设计（关系模型）

逻辑结构设计阶段的任务是将 E-R 图转换为特定数据库管理系统所支持的关系模型。对于关系型数据库 PostgreSQL，这意味着将实体和关系转换为二维表（Table）的结构，并定义主键（Primary Key）和外键（Foreign Key）来维护数据的一致性和完整性。

转换规则如下：

- 每个实体转换为一个独立的关系表。
- 实体的属性转换为表的列。
- 实体的主键作为表的主键。
- 一对多关系通过在“多”端实体对应的表中添加一个外键，该外键指向“一”端实体表的主键来实现。

根据此规则，我们得到以下三个关系表：

1. **user 表 (用户信息表)**

   该表由“用户”实体转换而来，用于存储所有注册用户的信息。

表 3-1：user 表结构

列名 (Column)	数据类型 (Type)	约束 (Constraint)	描述 (Description)
id	INTEGER	PRIMARY KEY, AUTO_INCREMENT	用户唯一标识符，主键
username	VARCHAR(80)	UNIQUE, NOT NULL	用户名，唯一且非空
email	VARCHAR(120)	UNIQUE, NOT NULL	用户邮箱，唯一且非空
password_hash	VARCHAR(255)	NOT NULL	存储加密后的密码哈希

2. **post 表 (帖子信息表)**

   该表由“帖子”实体转换而来，并通过外键 user_id 实现了与 user 表的一对多关系。

表 3-2：post 表结构

列名 (Column)	数据类型 (Type)	约束 (Constraint)	描述 (Description)
id	INTEGER	PRIMARY KEY, AUTO_INCREMENT	帖子唯一标识符，主键
title	VARCHAR(100)	NOT NULL	帖子标题
content	TEXT	NOT NULL	帖子正文内容
timestamp	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP, INDEX	发布时间，带索引以优化排序
user_id	INTEGER	FOREIGN KEY (references user.id), NOT NULL	作者ID，关联 user 表

3. **comment 表 (评论信息表)**

该表由“评论”实体转换而来，通过外键 user_id 和 post_id 分别实现了与 user 表和 post 表的一对多关系。

表 3-3：comment 表结构

列名 (Column)	数据类型 (Type)	约束 (Constraint)	描述 (Description)
id	INTEGER	PRIMARY KEY, AUTO_INCREMENT	评论唯一标识符，主键
content	TEXT	NOT NULL	评论正文内容
timestamp	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP, INDEX	发布时间，带索引以优化排序
user_id	INTEGER	FOREIGN KEY (references user.id), NOT NULL	评论者ID，关联 user 表
post_id	INTEGER	FOREIGN KEY (references post.id), NOT NULL	所属帖子ID，关联 post 表

3.3 物理结构设计

物理结构设计关注于数据在存储介质上的具体组织方式，包括数据类型的选择、索引的建立等，以达到最优的空间利用率和查询性能。本项目基于 PostgreSQL 数据库管理系统进行物理设计。

1. **数据类型选择：**

我们为逻辑模型中的通用数据类型选择了 PostgreSQL 中对应的高效物理类型：

整型 (INTEGER): 用于存储 id, user_id, post_id 等标识符。它占用空间小，索引效率高。

可变长度字符串 (VARCHAR(n)): 用于存储 username, email, password_hash, title 等有长度限制的字符串。相比 CHAR 类型，它能节约存储空间。

文本 (TEXT): 用于存储 content 字段，适用于长度不定的长文本内容，如帖子正文和评论。

时间戳 (TIMESTAMP): 用于存储 timestamp 字段，精确记录创建时间，便于按时间进行排序和筛选。

2. **索引策略：**

索引是提升数据库查询性能的关键。除了 PostgreSQL 自动为主键和唯一约束创建的索引外，我们还明确为以下字段设计了索引：

post.timestamp 和 comment.timestamp: 论坛的核心功能是按时间线展示内容，为时间戳字段创建索引能极大地加速 ORDER BY 排序操作。

外键字段 (user_id, post_id): 虽然 PostgreSQL 不会自动为外键创建索引，但为它们创建索引是最佳实践。这能显著提升 JOIN 操作和基于外键的 WHERE 查询（如“查询某用户的所有帖子”、“查询某帖子的所有评论”）的性能。在本项目中，Flask-SQLAlchemy 通常会自动为外键创建索引。

第四章：数据库实施

本章将详细介绍如何将第三章设计的数据库模型，通过具体的工具和代码在 PostgreSQL 环境中进行物理实现。

4.1 数据库环境搭建

数据库实施的第一步是构建一个稳定可靠的数据库运行环境。

1. **PostgreSQL 安装与配置:**
   我们选择在 Windows 操作系统上安装 PostgreSQL 11 版本。通过官方提供的图形化安装向导，完成了数据库服务、命令行工具以及 pgAdmin 4 图形化管理工具的安装。在安装过程中，我们设置了超级用户 postgres 的密码，并保持了默认的 5432 端口配置。
2. **创建项目数据库:**
   安装完成后，我们启动 pgAdmin 4 工具并连接到本地数据库服务。通过其图形化界面，我们新建了一个名为 flask_vue_forum 的数据库，其所有者为 postgres。这个空数据库将作为我们论坛应用的数据存储载体。

4.2 ORM 模型定义

为了在应用程序中以更高级、更面向对象的方式操作数据库，我们采用了对象关系映射（ORM）技术。Flask-SQLAlchemy 扩展使得我们能够用 Python 类来精确地定义和映射数据库中的表结构。

以下代码展示了 backend/app/models.py 文件中，如何将上一章设计的 user, post, comment 关系模型转换为 Python 代码。每个类对应一张表，类属性对应表的列，而 db.relationship 则清晰地定义了表之间的“一对多”关系。

backend/app/models.py 代码

```python
# app/models.py
from .extensions import db
from werkzeug.security import generate_password_hash,
check_password_hash
import datetime

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True,
nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password_hash = db.Column(db.String(255)) #此处由 128 增长为 255
    # --- 建立关系 ---
    # backref='author' 会在 Post 模型中创建一个虚拟的 .author 属性，可
以直接访问到关联的 User 对象
    posts = db.relationship('Post', backref='author', lazy=True)
```

```
    comments = db.relationship('Comment', backref='author',
lazy=True)

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)

# --- 新增 Post 模型 ---
class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    content = db.Column(db.Text, nullable=False)
    timestamp = db.Column(db.DateTime, index=True,
default=datetime.datetime.utcnow)
    # --- 建立外键关系 ---
    # user.id 指的是 user 表的 id 列
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=False)

    comments = db.relationship('Comment', backref='post',
lazy=True, cascade="all, delete-orphan")

    # 增加一个 to_dict 方法，方便序列化为 JSON
    def to_dict(self):
        return {
            'id': self.id,
            'title': self.title,
            'content': self.content,
            'timestamp': self.timestamp.isoformat() + 'Z',
            'user_id': self.user_id,
            'author': self.author.username, # 通过 backref 直接获取作
者名
        }


# --- 新增 Comment 模型 ---
class Comment(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    content = db.Column(db.Text, nullable=False)
    timestamp = db.Column(db.DateTime, index=True,
default=datetime.datetime.utcnow)
```

```
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'),
nullable=False)
    post_id = db.Column(db.Integer, db.ForeignKey('post.id'),
nullable=False)

    def to_dict(self):
        return {
            'id': self.id,
            'content': self.content,
            'timestamp': self.timestamp.isoformat() + 'Z',
            'user_id': self.user_id,
            'author': self.author.username,
            'post_id': self.post_id
        }
```

4.3 数据库迁移

在软件开发过程中，数据库模型会随着需求的变更而频繁演进。手动管理这些结构变更（Schema Changes）既繁琐又容易出错。为此，我们引入了 Flask-Migrate 工具，它能够自动化、版本化地管理数据库结构。

Flask-Migrate 基于强大的 Alembic 库，其核心思想是“代码即真理”。它将 Python 模型文件 (models.py) 视为数据库结构的唯一真实来源，通过比较模型与当前数据库的差异，来自动生成 SQL DDL（数据定义语言）脚本。

我们的实施流程如下：

1. **初始化迁移环境 (仅需一次):**

   在项目后端目录下，执行 flask db init。该命令会创建一个 migrations 文件夹，用于存放所有的迁移脚本和配置，这是数据库版本控制的起点。

2. **生成迁移脚本:**

   每当 models.py 中的模型发生变化(例如，新增一张表或一个字段)，我们执行 flask db migrate -m "描述性信息"。该命令会自动检测模型与数据库结构的差异，并生成一个新的 Python 迁移脚本，存放在 migrations/versions/ 目录下。这个脚本包含了将数据库升级到当前模型状态所需的 op.create_table()、op.add_column() 等 Alembic 操作。

3. **应用迁移:**

   生成迁移脚本后，我们执行 flask db upgrade。该命令会运行所有尚未应用的迁移脚本，将 SQL DDL 语句真正地应用到 PostgreSQL 数据库中，完成表的创建、修改或删除。在我们项目中，正是通过这个命令，在空的 flask_vue_forum 数据库中成功创建了 user, post, comment 等所有表。反之，flask db downgrade 命令则可以用于回滚到上一个版本。

通过这一套标准化的流程，我们实现了数据库结构的程序化、可重复和可追溯的管理，极大地提升了开发效率和系统的稳定性。

第五章：数据库应用系统开发

在完成了数据库的设计与实施之后，本章将重点阐述如何围绕该数据库构建一个完整的应用程序。遵循前后端分离的设计思想，我们将应用系统分为后端 API 服务和前端用户界面两大部分，并详细介绍其关键功能的实现逻辑。

5.1 后端 API 开发

后端系统的核心任务是作为前端应用与数据库之间的安全、高效的桥梁。为此，我们采用轻量级的 Python Web 框架 Flask 来构建一套符合 RESTful (Representational State Transfer) 风格的 API 接口。RESTful API 将系统中的每一个资源（如用户、帖子）抽象为一个唯一的 URI (Uniform Resource Identifier)，并使用标准的 HTTP 方法（GET, POST, PUT, DELETE）来定义对这些资源的操作，实现了接口的清晰性、无状态性和统一性。

在本项目中，后端服务接收前端通过 HTTP 协议发送的 JSON 格式数据，解析请求后，利用 Flask-SQLAlchemy 这个 ORM 工具执行相应的数据库操作（增、删、改、查）。完成操作后，再将结果（无论是成功信息还是数据对象）序列化为 JSON 格式返回给前端。此外，通过 Flask-JWT-Extended 扩展，我们为需要权限的接口（如创建帖子）添加了基于 Token 的认证保护，确保了数据操作的安全性。

以下是系统中几个核心 API 接口的实现代码及其功能解析。

1. **用户注册接口**

   此接口负责处理新用户的注册请求。它是一个公开的 POST 接口，接收包含用户名、邮箱和密码的 JSON 数据。

   功能说明：

   接口首先从请求体中获取用户提交的数据。为了保证数据的唯一性，它会先查询数据库，检查用户名和邮箱是否已被注册。若数据有效，则创建一个新的 User 模型实例，调用 set_password 方法对明文密码进行哈希加密，然后将新用户对象添加到数据库会话中，并通过 db.session.commit() 将其持久化到 user 表中。

用户注册接口代码片段(backend/app/api/auth.py)

```
@auth_bp.route('/register', methods=['POST'])
def register():
    data = request.get_json()
    username = data.get('username')
    email = data.get('email')
    password = data.get('password')

    if not username or not email or not password:
        return jsonify({'message': 'Missing username, email, or
password'}), 400

    if User.query.filter_by(username=username).first() is not
None:
        return jsonify({'message': 'Username already exists'}),
409
```

```
    if User.query.filter_by(email=email).first() is not None:
        return jsonify({'message': 'Email already exists'}),
409

    new_user = User(username=username, email=email)
    new_user.set_password(password)
    db.session.add(new_user)
    db.session.commit()

    return jsonify({'message': 'User created successfully'}),
201
```

2. **用户登录接口**

此接口负责验证用户身份并签发访问令牌（Access Token）。

功能说明：

接口接收包含用户名和密码的 POST 请求。它首先根据用户名查询 user 表。如果用户存在，则调用 check_password 方法，将用户提交的明文密码与数据库中存储的哈希值进行比对。验证通过后，使用 flask_jwt_extended 的 create_access_token 函数，以用户 ID 为身份标识（identity），生成一个 JWT，并将其返回给前端。

用户登录接口代码片段(backend/app/api/auth.py)

```
@auth_bp.route('/login', methods=['POST'])
def login():
    data = request.get_json()
    username = data.get('username')
    password = data.get('password')

    user = User.query.filter_by(username=username).first()

    if user is None or not user.check_password(password):
        return jsonify({'message': 'Invalid username or
password'}), 401

    access_token = create_access_token(identity=str(user.id))
    return jsonify(access_token=access_token)
```

3. **获取帖子列表接口**

此接口是论坛首页的数据来源，负责返回所有帖子的列表。

功能说明：

这是一个公开的 GET 接口。它通过 Post.query 查询 post 表中的所有记录，并使用 .order_by(Post.timestamp.desc()) 将结果按发布时间进行降序排序，以确保最新的帖子显示在最前面。最后，它遍历查询结果，调用每个 post 对象的 to_dict() 方法将其转换为字典，最终将字典列表序列化为 JSON 数组返回。

获取帖子列表接口代码片段(backend/app/api/posts.py)

```
@posts_bp.route('/<int:id>', methods=['GET'])
def get_post(id):
    post = Post.query.get_or_404(id)
    return jsonify(post.to_dict())
```

4. **创建帖子接口**

此接口用于处理登录用户发布新帖子的请求，是典型的带认证的写入操作。

功能说明：

该接口使用 @jwt_required() 装饰器进行保护，意味着只有在请求头中包含了有效 JWT 的请求才能访问。接口通过 get_jwt_identity() 从 Token 中安全地获取当前登录用户的 ID。然后，它根据用户提交的标题、内容以及获取到的用户 ID 创建一个新的 Post 模型实例，并将其存入数据库。

创建帖子接口代码片段(backend/app/api/posts.py)

```
@posts_bp.route('/', methods=['POST'])
@jwt_required()
def create_post():
    data = request.get_json()
    current_user_id = get_jwt_identity()

    post = Post(
        title=data['title'],
        content=data['content'],
        user_id=current_user_id
    )
    db.session.add(post)
    db.session.commit()

    return jsonify(post.to_dict()), 201
```

5.2 前端界面开发

前端系统采用 Vue.js 3 框架构建为一个单页应用（SPA）。SPA 的核心优势在于其出色的用户体验：应用的初始化过程仅需加载一次核心的 HTML、CSS 和 JavaScript 文件，后续的所有页面切换和数据交互都通过 JavaScript 在客户端动态完成，无需重新向服务器请求整个页面。

我们通过组件化的思想来构建界面，将页面拆分为多个可复用的组件。使用 Vue Router 管理应用的路由，实现页面间的流畅跳转。通过 Vuex 进行全局状态管理，集中存放如用户登录状态、Token 等共享数据。数据交互则完全依赖于 Axios 库，它负责异步调用后端提供的 RESTful API，获取数据后，利用 Vue 的响应式系统将数据动态地渲染到视图上，实现了数据与视图的绑定。

1. **登录/注册组件 (Login.vue, Register.vue)**

这两个组件负责收集用户的输入，并调用后端认证 API。

功能说明：

组件的 <template> 部分包含标准的 HTML 表单元素（如 input, button）。通过 Vue 的 v-model 指令，我们将表单输入框的值与组件 <script> 部分 data 对象中的 username, password 等变量进行双向绑定。当用户点击提交按钮时，会触发一个方法（如 handleLogin），该方法会收集 data 中的数据，然后调用 Vuex store 中封装好的 login 或 register action。这个 action 内部使用 Axios 将数据以 POST 请求发送到后端的 /api/auth/login 或 /api/auth/register 接口。同时，组件还包含错误处理逻辑，能够捕获 API 返回的错误信息并展示给用户。

登录/注册组件核心代码 (以 Login.vue 为例)

```
<template>
  <div class="login-container">
    <h2>登录</h2>
    <!-- .prevent 阻止表单默认提交行为 -->
    <form @submit.prevent="handleLogin">
      <div class="form-group">
        <label for="username">用户名</label>
        <input type="text" v-model="username" id="username"
required />
      </div>
      <div class="form-group">
        <label for="password">密码</label>
        <input type="password" v-model="password" id="password"
required />
      </div>
      <button type="submit">登录</button>
      <p v-if="errorMessage" class="error">{{ errorMessage }}</p>
    </form>
  </div>
</template>

<script>
import { mapActions } from "vuex";

export default {
  name: "LoginView",
  data() {
    return {
```

```
      username: "",
      password: "",
      errorMessage: "",
    };
  },
  methods: {
    // 使用展开运算符将 vuex 的 actions 混入到 methods 中
    ...mapActions(["login"]),

    async handleLogin() {
      try {
        const userData = {
          username: this.username,
          password: this.password,
        };
        // 调用映射过来的 login action
        await this.login(userData);
        // 登录成功后给用户反馈
        alert("登录成功！");
        // 之后可以跳转到首页，例如：
        // this.$router.push({ name: 'home' });
      } catch (error) {
        // 从后端响应中获取更友好的错误信息
        this.errorMessage =
          error.response?.data?.message || "登录失败，请检查您的用户名和密码。";
      }
    },
  },
};
</script>

<style scoped>
.login-container {
  max-width: 400px;
  margin: 50px auto;
  padding: 20px;
  border: 1px solid #ccc;
  border-radius: 5px;
}

.form-group {
  margin-bottom: 15px;
}
```

```
label {
  display: block;
  margin-bottom: 5px;
}

input {
  width: 100%;
  padding: 8px;
  box-sizing: border-box;
}

button {
  width: 100%;
  padding: 10px;
  background-color: #007bff;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
}

.error {
  color: red;
  margin-top: 10px;
}
</style>
```

2. **首页组件 (Home.vue)**

该组件是论坛的入口，负责展示所有帖子的列表。

功能说明：

该组件利用 Vue 的生命周期钩子 created()。当组件实例被创建时，它会自动调用 Vuex store 中的 fetchPosts action。该 action 会向后端 /api/posts/ 接口发送一个 GET 请求。获取到帖子数据列表后，数据被存入 Vuex 的全局 state。Home.vue 组件通过 mapState 辅助函数将这个 state 映射为自己的计算属性 posts。在模板中，使用 v-for 指令遍历 posts 数组，为每一篇帖子动态渲染出一个包含标题、作者和时间的列表项。每个列表项都是一个 router-link，可以点击跳转到对应的帖子详情页。

首页组件核心代码(Home.vue)

```
<template>
  <div class="home-container">
    <div class="header">
      <h1>论坛首页</h1>
      <!-- 只有登录后才显示此按钮 -->
```

```
      <router-link v-if="isLoggedIn" :to="{ name: 'createPost' }">
        <button>发布新帖子</button>
      </router-link>
    </div>

    <div v-if="posts.length">
      <div v-for="post in posts" :key="post.id" class="post-item">
        <!-- 使用 router-link 跳转到帖子详情页 -->
        <router-link :to="{ name: 'postDetail', params: { id:
post.id } }">
          <h2>{{ post.title }}</h2>
        </router-link>
        <p>
          作者: {{ post.author }} | 发布于:
          {{ new Date(post.timestamp).toLocaleString() }}
        </p>
      </div>
    </div>
    <div v-else>
      <p>正在加载帖子... 或 还没有任何帖子。</p>
    </div>
  </div>
</template>

<script>
// --- 修复点 1: 导入 mapGetters ---
import { mapState, mapActions, mapGetters } from "vuex";

export default {
  name: "HomeView",
  computed: {
    // 使用 mapState 辅助函数将 store 中的 posts 映射到组件的 computed
属性
    ...mapState(["posts"]),
    // --- 修复点 2: 映射 isLoggedIn getter ---
    ...mapGetters(["isLoggedIn"]),
  },
  methods: {
    ...mapActions(["fetchPosts"]),
  },
  created() {
    // 当组件被创建时，调用 action 来获取帖子列表
    this.fetchPosts();
  },
```

```
};
</script>

<style scoped>
.home-container {
  max-width: 800px;
  margin: 20px auto;
}

/* --- 修复点 3: 添加 header 样式 --- */
.header {
  display: flex;
  justify-content: space-between;
  align-items: center;
  margin-bottom: 20px;
}
.header h1 {
  margin: 0;
}
/* ---------------------------------- */

.post-item {
  border: 1px solid #eee;
  padding: 15px;
  margin-bottom: 15px;
  border-radius: 5px;
  text-align: left;
}
.post-item h2 {
  margin: 0 0 10px 0;
}
.post-item p {
  color: #666;
  font-size: 0.9em;
}
.post-item a {
  text-decoration: none;
  color: #2c3e50;
}
.post-item a:hover {
  text-decoration: underline;
}
</style>
```

3. **帖子详情页**

该组件负责展示单篇帖子的完整内容以及其下方的所有评论。

功能说明：

该组件的设计利用了 Vue Router 的动态路由功能。路由配置为 /post/:id，其中 :id 是一个动态参数。当用户访问例如 /post/1 时，PostDetail.vue 组件会通过 props 接收到 id=1 这个参数。在组件的 created() 生命周期钩子中，它会使用这个 id 同时调用 Vuex store 中的 fetchPost(id) 和 fetchComments(id) 两个 action，分别向后端的 /api/posts/1 和 /api/posts/1/comments 接口请求数据。获取到数据后，同样存入 Vuex state 并映射到组件的计算属性 post 和 comments，最终动态地渲染出帖子详情和评论列表。

帖子详情页核心代码 (PostDetail.vue)

```vue
<!-- 模板部分：列表渲染 -->
<template>
  <div v-for="post in posts" :key="post.id">
    <router-link :to="{ name: 'postDetail', params: { id:
post.id } }">
      <h2>{{ post.title }}</h2>
      <p>作者: {{ post.author }}</p>
    </router-link>
  </div>
</template>

<!-- 脚本部分：数据获取与映射 -->
<script>
import { mapState, mapActions } from "vuex";

export default {
  computed: {
    ...mapState(["posts"]), // 映射全局 state.posts 到 this.posts
  },
  methods: {
    ...mapActions(["fetchPosts"]), // 映射 action
  },
  created() {
    // 组件创建时自动获取数据
    this.fetchPosts();
  },
};
</script>
```

第六章：系统运行与演示

本章将对已完成的论坛数据库应用系统进行运行演示。首先简要概述将应用部署到云平台，使其能够通过公网访问的过程，然后通过一系列界面截图，详细展示系统的各项核心功能，包括用户认证、内容浏览、信息发布与权限控制等。

6.1 系统部署概述

为了验证本系统在真实网络环境下的可用性，并完整地模拟一个线上产品的生命周期，我们选择将前后端应用分别部署到现代化的云平台。这种部署方式不仅体现了前后端分离架构的优势，也实践了持续集成/持续部署（CI/CD）的工程理念。

1. **后端部署 (Render):**

平台选择： 我们选择了对 Python/Flask 项目支持友好的 PaaS（平台即服务）平台 Render.com。

部署流程：

我们将后端代码推送到 GitHub 仓库。

在 Render 平台创建一个新的 Web Service，并关联该 GitHub 仓库，同时指定根目录为 backend/。

配置构建命令为 pip install -r requirements.txt，启动命令为 flask db upgrade && gunicorn "app:create_app()"。此启动命令巧妙地集成了数据库自动迁移功能，确保每次部署时数据库结构都能与代码模型保持同步。

在 Render 上创建一个免费的 PostgreSQL 数据库实例，并将其 Internal Database URL 配置为后端服务的环境变量 DATABASE_URL。同时，SECRET_KEY, JWT_SECRET_KEY 等敏感信息也通过环境变量注入，保证了代码的安全性。

最后，将前端部署后生成的 Vercel URL 配置为环境变量 FRONTEND_URL，以实现生产环境下的 CORS 跨域许可。

成果： 部署成功后，我们获得了一个稳定、可通过公网访问的后端 API 服务地址 https://flask-vue-forum-api.onrender.com （免费版试用期即将结束，点击后可能 404）.

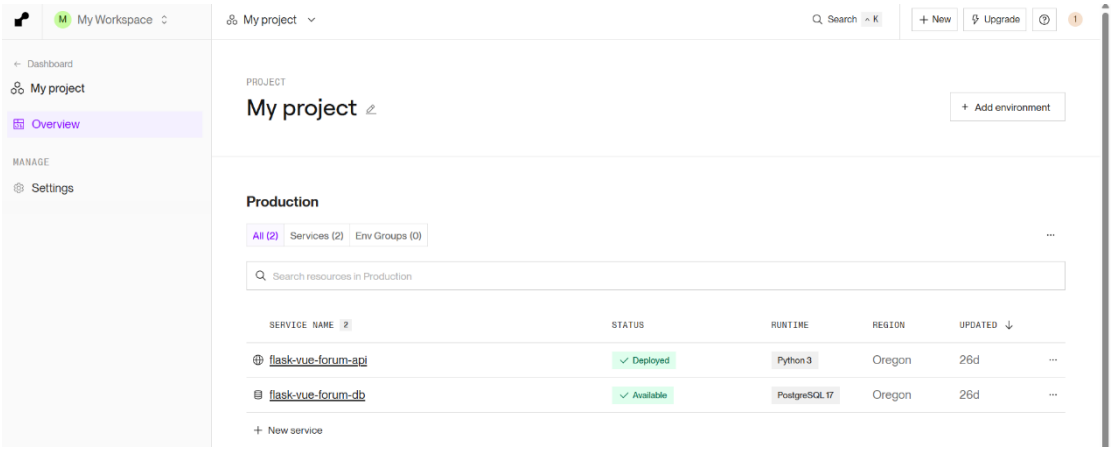

图 6- 1 render.com 后端部署

2. **前端部署 (Vercel):**

平台选择： 我们选择了业界领先的静态网站托管平台 Vercel.com。

部署流程：

Vercel 与我们的 GitHub 仓库直接集成。

我们创建了 .env.production 文件，在其中将 VUE_APP_API_BASE_URL 指向我们部署在 Render 上的后端 API 地址。

在 Vercel 平台创建一个新项目，关联 GitHub 仓库并指定根目录为 frontend/。

Vercel 能够智能识别出这是一个 Vue CLI 项目，自动配置了构建命令 (npm run build) 和输出目录。

成果： 每当我们向 GitHub 的主分支推送新的前端代码，Vercel 都会自动拉取、构建并部署，整个过程完全自动化。部署成功后，我们获得了一个全球 CDN 加速的前端应用访问地址 https://flask-vue-forum.vercel.app/（免费版使用权即将结束，点击后可能只有前端渲染效果，接收不到后端数据）。

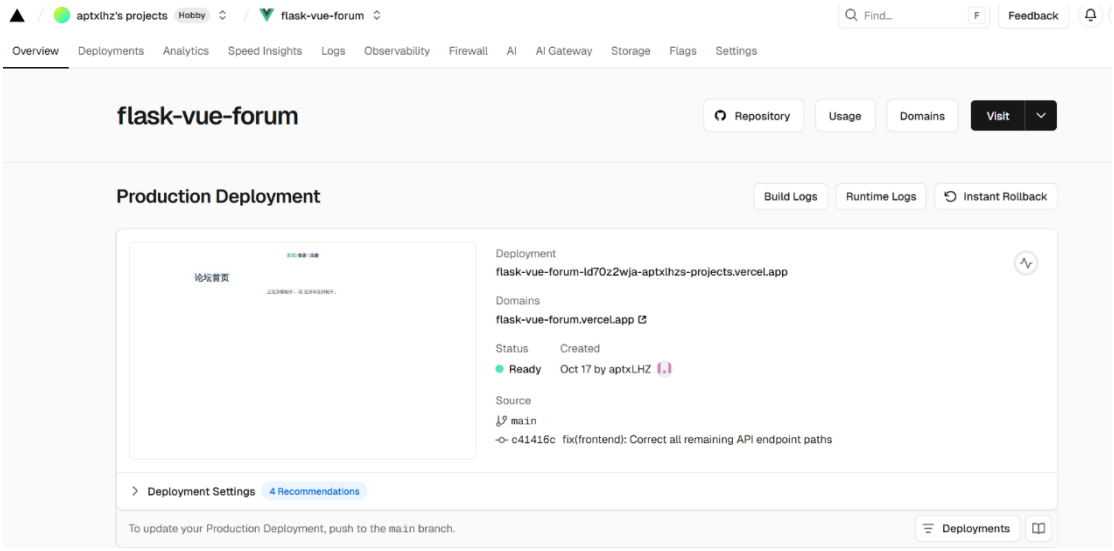

图 6- 2 vercel.com 前端部署

通过以上部署流程，我们成功地将本地开发的应用发布为可供公众使用的线上产品，验证了项目架构的完整性和健壮性。

6.2 核心功能演示

以下将通过一系列系统运行截图，来演示本论坛系统的主要功能模块。

1. **用户注册与登录**

系统提供了安全、独立的用户认证流程。用户首先需要注册一个账户，然后才能登录以使用发帖、评论等核心功能。

图 6- 4 用户注册界面

如图 6-3，用户在此页面输入唯一的用户名、邮箱和密码，点击“注册”按钮。前端将对输入进行基本校验，然后将数据发送至后端 /api/auth/register 接口。后端完成数据验证和入库操作后，前端会提示用户注册成功并引导至登录页面。

图 6- 3 用户登录页面

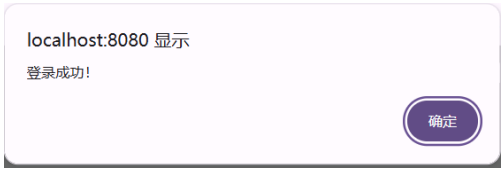

图 6- 5 登录成功提示

如图 6-4，已注册用户在此页面输入正确的凭据进行登录。前端将数据发送至 /api/auth/login 接口。后端验证成功后，返回一个 JWT Token，并提示登陆成功，如图 6-5 所示。

图 6- 6 登录成功后的界面变化

如图 6-6，前端在接收到 JWT Token 后，会将其存储在本地（localStorage），并更新全局状态。界面的导航栏会发生动态变化，原先的“登录”、“注册”链接变为“登出”链接，同时“发布新帖子”等需要登录权限的按钮也会随之出现，表明用户已成功进入登录状态。

2. **帖子浏览**

论坛的首页是内容的核心展示区，所有用户均可在此浏览帖子列表。

图 6-7 论坛首页帖子列表

如图 6-7，首页组件在加载时，会自动向后端 /api/posts/ 接口请求数据。后端返回按时间降序排列的帖子列表，前端通过 v-for 指令动态渲染出每一篇帖子的摘要信息。列表中的每个标题都是一个超链接，点击后可进入帖子详情页。

3. **发布与评论**

登录用户可以积极参与社区互动，包括发布新话题和对他人的帖子进行评论。

图 6-8 创建新帖子页面

如图 6-8，登录用户点击首页的“发布新帖子”按钮后，会跳转到此页面。用户填写标题和内容，提交后，前端将数据 POST 到受保护的 /api/posts/ 接口。发布成功后，页面

会自动跳转回首页，并显示成功的全局通知。

图 6- 9 帖子详情页及评论功能

如图 6-9， 在帖子详情页，用户可以看到帖子的完整内容。下方为评论区，登录用户可以看到评论输入框，输入内容后即可发表评论。评论成功后，评论列表会进行局部刷新，新评论会即时出现在列表末尾，整个过程无需刷新页面。

4. **权限验证**

系统通过前后端双重校验，确保用户只能对自己发布的内容进行管理操作，例如删除。

如图 6-9，当登录用户浏览自己发布的帖子时，前端通过解析 JWT Token 获取当前用户 ID，并与帖子的作者 ID 进行比对。如果两者一致，页面上会渲染出“删除帖子”按钮。

图 6- 10 非作者视角下的帖子详情页

如图 6-10，当用户浏览他人发布的帖子时，由于用户 ID 与作者 ID 不匹配，v-if 条件判断为假，“删除帖子”按钮不会被渲染出来。这从用户界面层面防止了越权操作的可能。同时，即使有人恶意地通过技术手段模拟删除请求，后端的 API 接口也会进行二次身份校验，从根本上保证了数据的安全性。

第七章：总结与展望

7.1 项目总结

本次项目成功地完成了一个从零到一的、基于前后端分离架构的现代化论坛数据库系统的设计与开发。我们完整地经历了从项目选题、需求分析，到数据库的概念、逻辑、物理设计，再到基于 PostgreSQL 的数据库实施，并最终开发了功能完备的后端 API 服务和前端用户界面，实现了项目的既定目标。系统目前已具备用户注册登录、帖子发布删除、实时评论等核心功能，并通过 JWT 实现了安全的认证授权。

通过本次项目实践，我获得了宝贵的综合性知识和技能，主要体现在以下几个方面：

数据库设计与实施： 深入理解并实践了数据库设计的范式理论，能够独立完成从 E-R 图到关系模型的转换，并利用 ORM 和数据库迁移工具（Flask-SQLAlchemy, Flask-Migrate）高效、规范地进行数据库的物理实现和版本管理。

全栈开发能力：掌握了前后端分离架构的开发模式。后端层面，能够使用 Flask 框架构建符合 RESTful 规范的数据 API；前端层面，能够运用 Vue.js 框架构建动态、响应式的用户界面，并熟练处理组件化、路由和状态管理等问题。

项目管理与工程化思维：学会了使用 Git 进行版本控制，理解了模块化、组件化的代码组织方式，并初步接触了 CI/CD 自动化部署流程，培养了从全局视角审视项目架构的工程化思维。

系统调试与问题解决：在开发过程中，独立分析并解决了 CORS 跨域、Token 认证（过期、权限）、数据库迁移、生产环境配置等一系列真实世界中常见的技术难题。通过阅读日志、分析网络请求，我的系统调试和独立解决问题的能力得到了极大的锻炼。

7.2 未来展望

本项目作为一个最小可行产品（MVP），已搭建了坚实的基础架构，同时也为未来的功能扩展留下了广阔的空间。后续可以从以下几个方面对系统进行进一步的优化和完善：

1、功能增强：

增加帖子编辑功能： 允许用户在发布帖子后对其内容进行修改。

实现图片上传与存储：支持用户在帖子和评论中上传图片，后端可集成云存储服务（如 AWS S3 或 Cloudinary）进行处理。

添加用户个人主页： 创建用户个人中心，展示用户发布的所有帖子和评论。

引入富文本编辑器： 将帖子和评论的输入框升级为富文本编辑器（如 TinyMCE 或 Quill.js），支持更丰富的文本格式。

2、性能优化：引入分页加载： 当帖子和评论数量巨大时，一次性加载所有数据会导致性能瓶颈。可以为帖子列表和评论列表接口添加分页功能，实现无限滚动或分页按钮加载。

3、用户体验提升：

实现 Refresh Token 机制： 引入 Refresh Token 来实现无感知的 Token 刷新，避免因 Access Token 过期而需要用户频繁重新登录，提升长期使用的流畅性。**实时通知系统：** 使用 WebSocket 或 SSE 技术，实现当有新评论时，对帖子作者进行实时消息推送。

4、代码与架构优化：

单元测试与集成测试： 为后端的关键业务逻辑编写单元测试，确保代码质量和重构的安全性。**容器化部署：**使用 Docker 将前后端应用容器化，实现更标准、更便捷的部署和环境迁移。

致谢

随着数据库实验课程的学习接近尾声，在本课程论文完成之际，我们谨向**老师致以最诚挚的谢意。

在整个学期的学习过程中，*老师凭借深厚的专业知识和深入浅出的讲解，为我们系统地构建了数据库实践的知识体系。从数据库的基本操作到索引等复杂技术，每一堂课都让我们受益匪。这门课程不仅传授了宝贵的专业知识，更重要的是激发了我们对数据库领域的浓厚兴趣。我们将带着这份由课程点燃的热情，在未来的学习和实践中继续探索数据库的无限可能。

再次感谢**老师的辛勤付出与悉心指导！