

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | B/S体系软件设计 |
| 实验名称： | 商品比价网站 |
| 姓 名： | 周玥儿 |
| 学 院： | 计算机学院 |
| 系： | 计算机科学与技术 |
| 专 业： | 计算机科学与技术 |
| 学 号： | 3220102179 |
| 指导教师： | 胡晓军 |

2024年 12月 23 日

## 1 项目概述

### 1.1 项目背景

在电子商务日益普及的背景下，用户对商品价格的比较和获取变得更加重要。此项目旨在开发一个比价系统，用户可以通过该系统查询主流电商平台的商品价格，获取历史价格走势，并在价格下降时接收提醒。

### 1.2 项目目标

构建一个能够查询商品价格、提供价格走势图和降价提醒的比价系统，支持用户注册与登录，并实现移动端友好显示。

### 1.3 功能需求

#### 1.3.1 用户注册与登录

* **用户注册**:
  + 填写必要信息（用户名、密码、邮箱等）
  + 验证信息有效性：
    - 用户名、密码长度至少为6字节
    - 邮箱格式验证
    - 确保用户名和邮箱在系统中唯一
* **用户登录**:
  + 验证用户名和密码的正确性
  + 登录后可访问系统的其他功能

#### 1.3.2 商品价格查询

* **通过商品名称查询**:
  + 分词处理优化查询
  + 支持查询多个结果
  + 支持至少两个平台（如淘宝、京东）的价格查询
  + 针对需要用户登录的电商平台，支持用户验证
  + 显示商品信息并提供历史价格走势图表

#### 1.3.3 商品库建立

* **数据库设计**:
  + 保存商品信息，包括名称、多级品类、规格、条码、图片等
  + 支持商品价格的历史记录

#### 1.3.4 降价提醒功能

* 定时查询指定商品的最新价格
* 提供降价提醒方式（邮件、App推送等）

#### 1.3.5 移动端适配

* 优化样式以适应手机浏览器、微信等应用内置浏览器
* 如果开发手机App，支持相机拍摄商品图片或扫码条码查询商品

### 1.4 技术架构

#### 1.4.1 前端技术栈

* Vue 3
* TypeScript
* Vite
* Pinia (状态管理)
* Naive UI (组件库)
* Axios (HTTP客户端)
* ECharts (图表可视化)
* Nginx（部署反向代理）

#### 1.4.2 后端技术栈

* Flask (Python Web框架)
* MySQL (数据库)
* Redis (缓存，可选)
* Celery (异步任务队列，用于价格监控)
* JWT (用户认证)

## 2 系统架构设计

### 2.1 整体架构

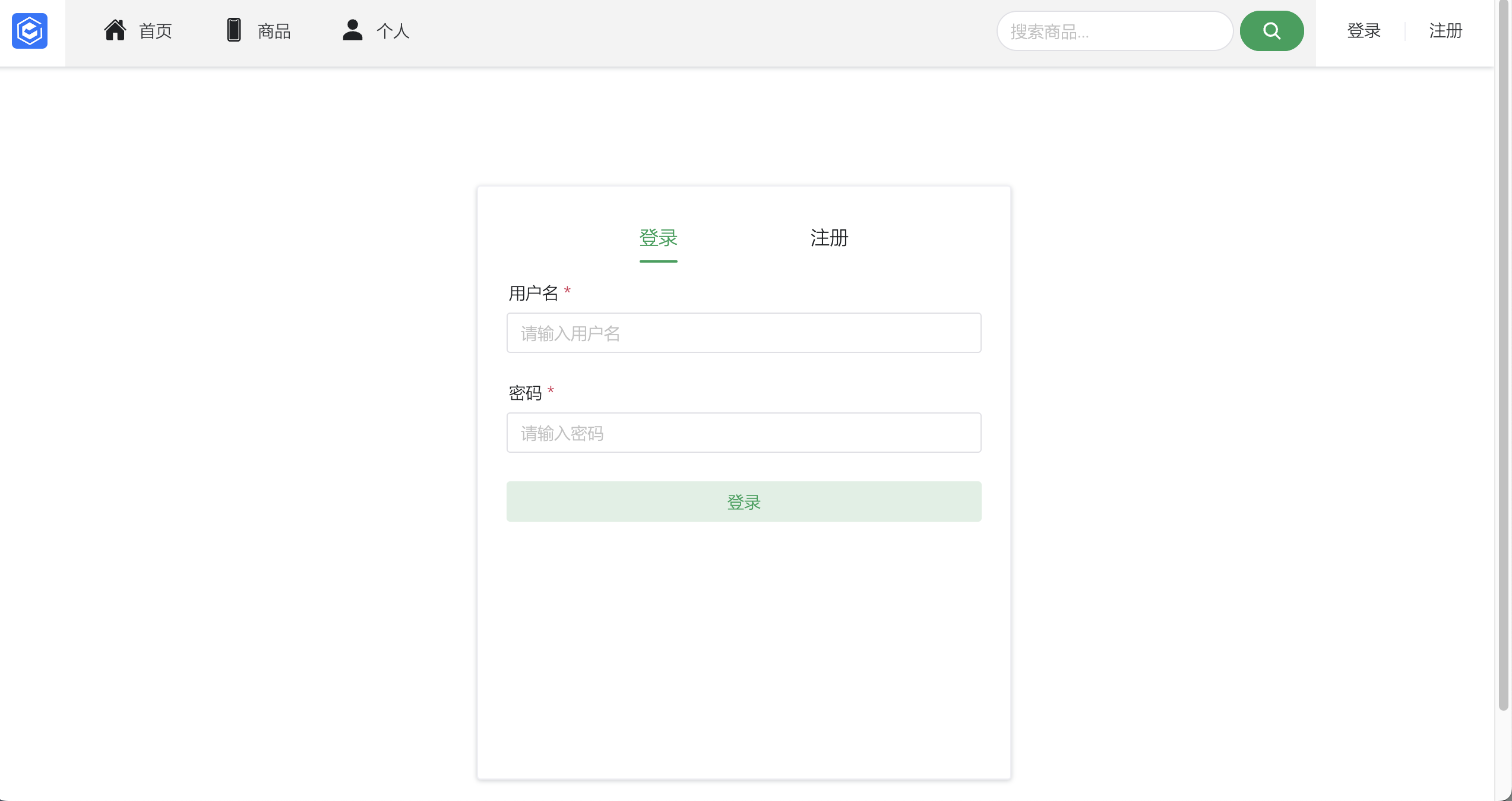
* 前端层：SPA应用 + 移动端适配
* 接口层：RESTful API
* 服务层：业务逻辑处理
* 数据层：MySQL存储 + Redis缓存
* 任务层：异步任务处理

### 2.2 系统模块划分

1. 用户认证模块
2. 商品搜索模块
3. 价格比较模块
4. 数据采集模块
5. 价格监控模块
6. 系统管理模块

## 3 功能模块设计

### 3.1 用户认证模块



#### 3.1.1 功能描述

* 用户注册：邮箱验证、用户名密码设置、基本信息填写
* 用户登录：账号密码登录、记住登录状态
* 密码重置：邮箱验证重置、安全问题验证
* 个人信息管理：基本信息修改、密码修改、通知设置

#### 3.1.2 核心组件设计

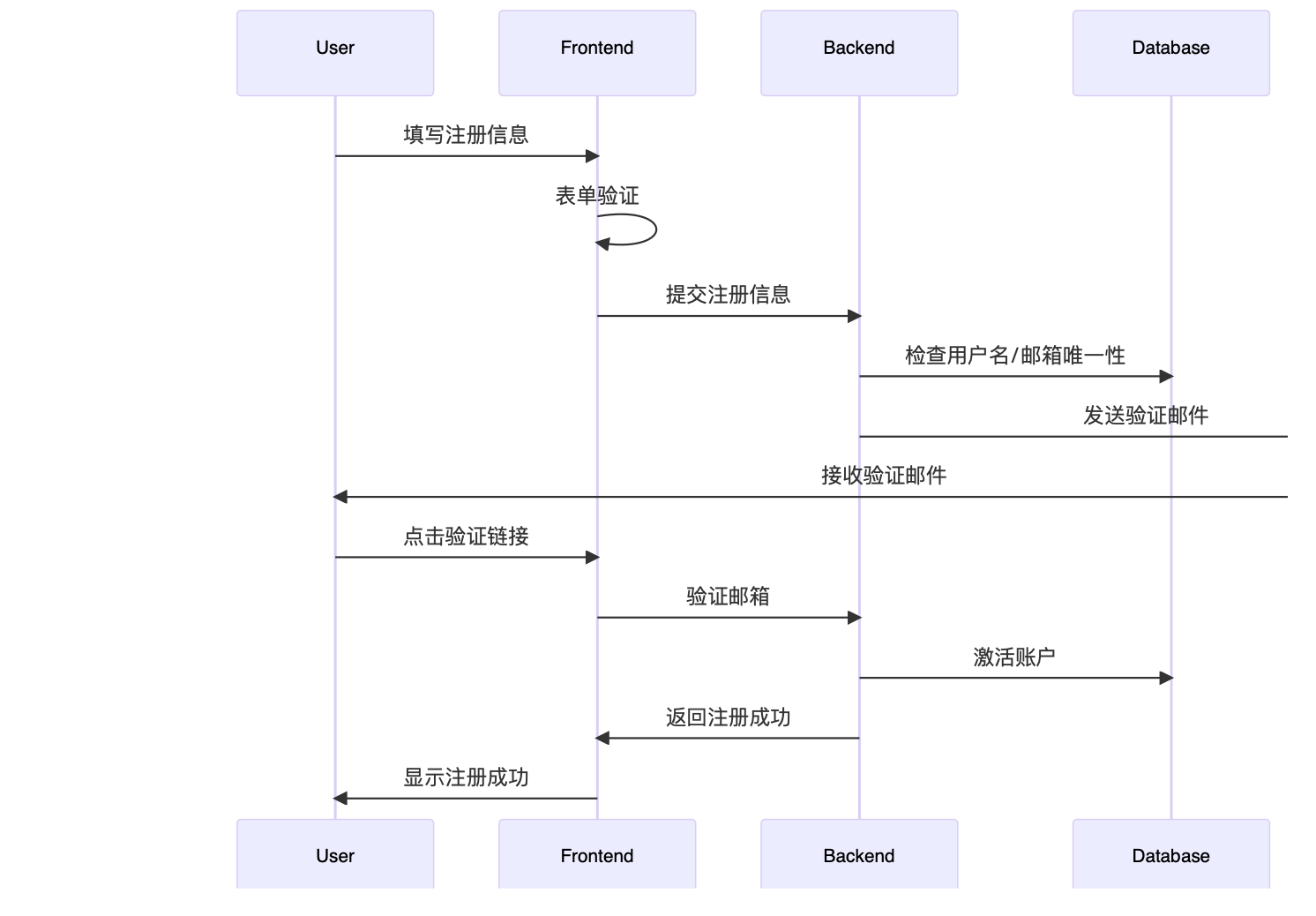
1. 用户认证页面(登录注册)

interface UserState {  
 isLoggedIn: boolean;  
 userInfo: {  
 id: number;  
 username: string;  
 email: string;  
 avatar?: string;  
 preferences: UserPreferences;  
 };  
 token: string;  
}

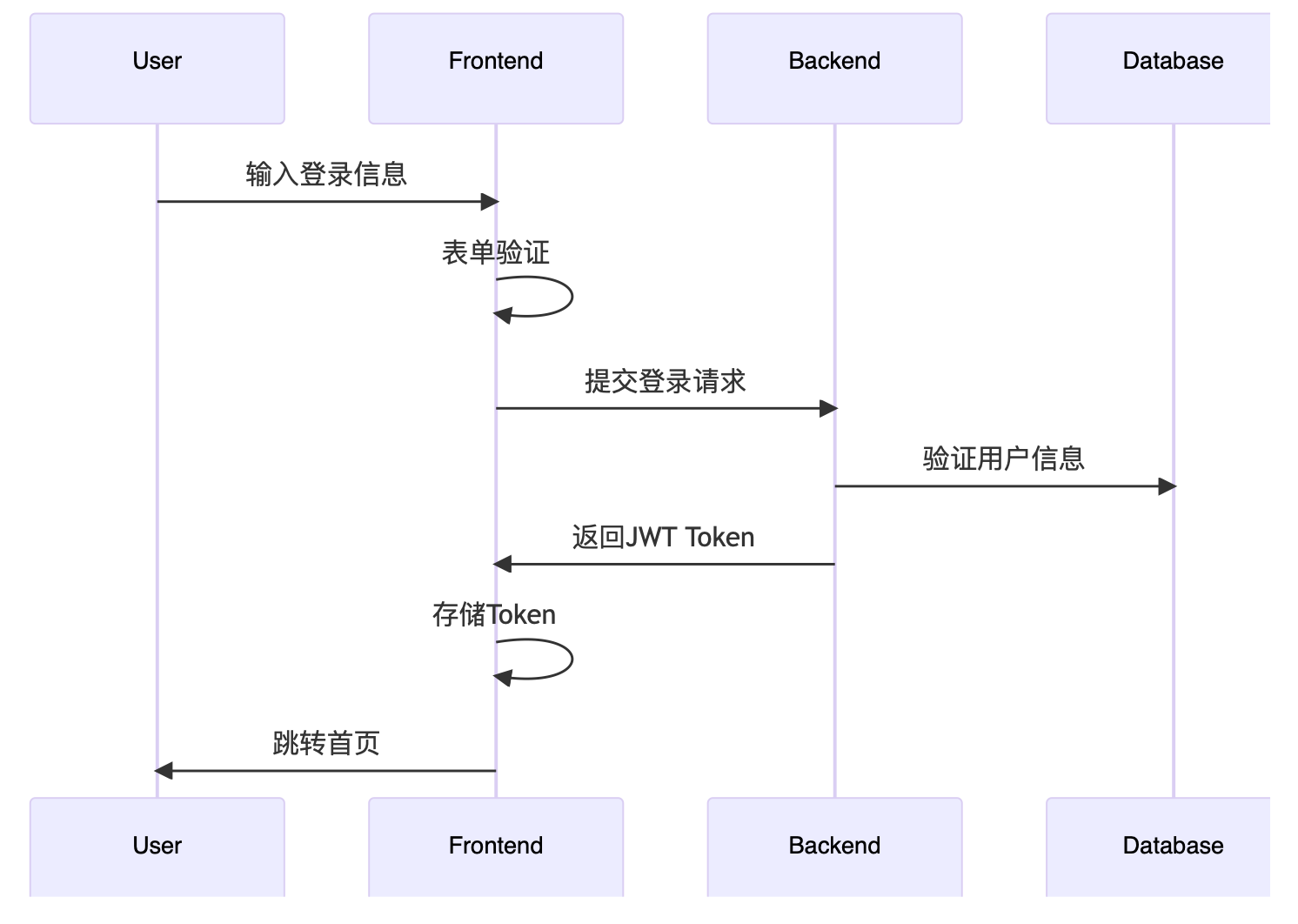
1. 用户信息页面
2. 状态管理 (LoginStore)

#### 3.1.3 功能实现流程

1. 用户注册流程



1. 登录认证流程



登陆流程

1. **用户登录**：用户通过登录接口提交用户名和密码。
2. **生成Token**：后端验证用户身份成功后，生成一个令牌。令牌通常包含用户信息、有效期、签名等。
3. **返回Token**：令牌返回给客户端，由客户端（如浏览器或移动应用）保存在localStorage中。

后续请求

1. **请求携带Token**：客户端每次请求时，将Token放在HTTP请求头（Authorization: Bearer <token>）中发送到服务器。
2. **Token验证**：后端验证Token的签名和有效期，确认用户身份。
3. **Token失效处理**：当Token过期后，用户需要重新登录以获取新的令牌，或通过刷新Token机制（如Refresh Token）来延长会话。

#### 3.1.4 安全考虑

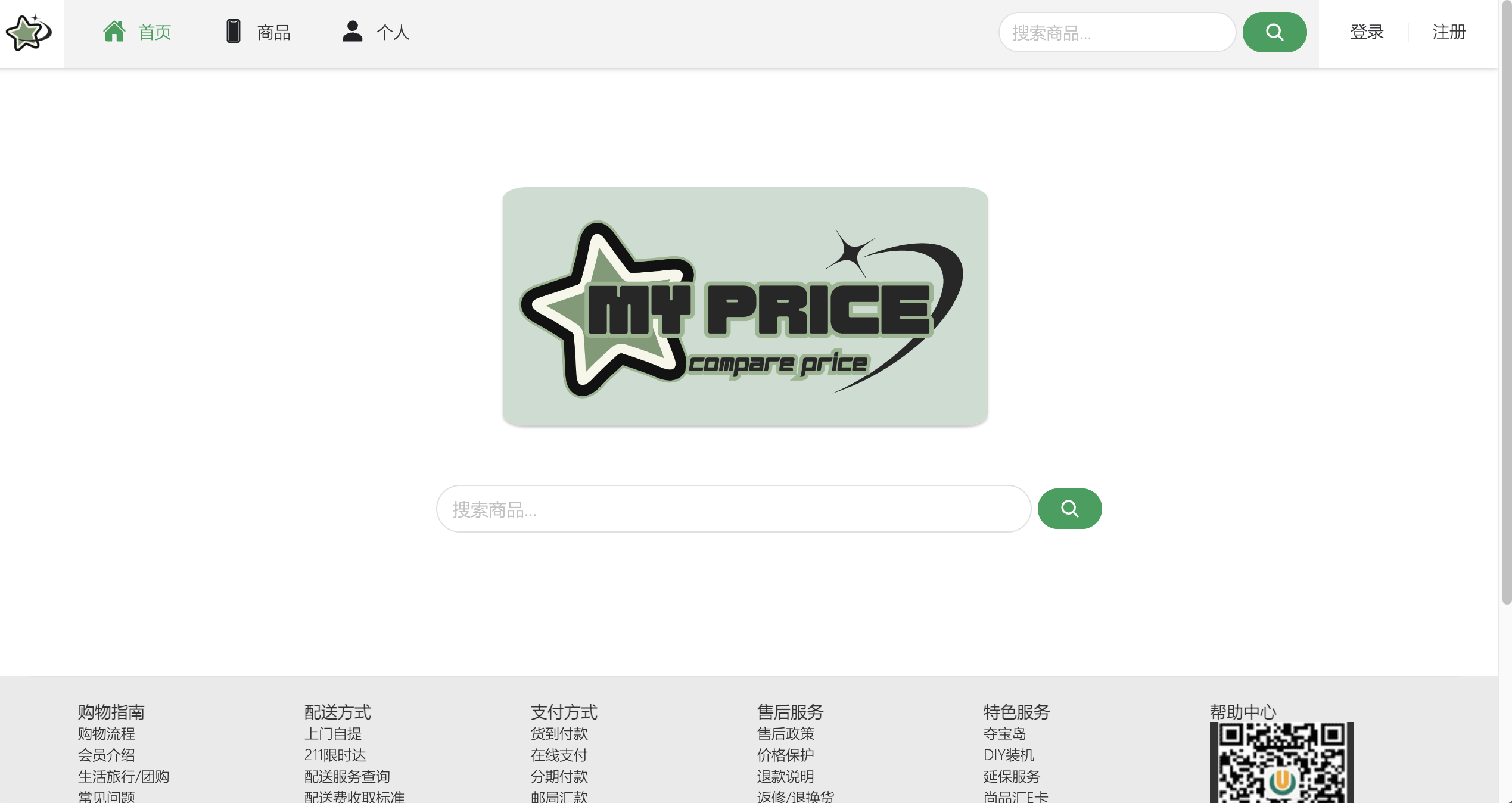
1. 密码安全

* 使用bcrypt进行密码加密存储
* 密码强度要求：最少6位，包含大小写字母和数字

1. 会话安全

* JWT Token存储在localStorage
* Token有效期为24小时
* 使用refresh token机制

### 3.2 商品搜索模块

* 未登录前的首页以搜索框的形式呈现
* 搜索结果展示（还需要添加筛选模块）

#### 3.2.1 功能描述

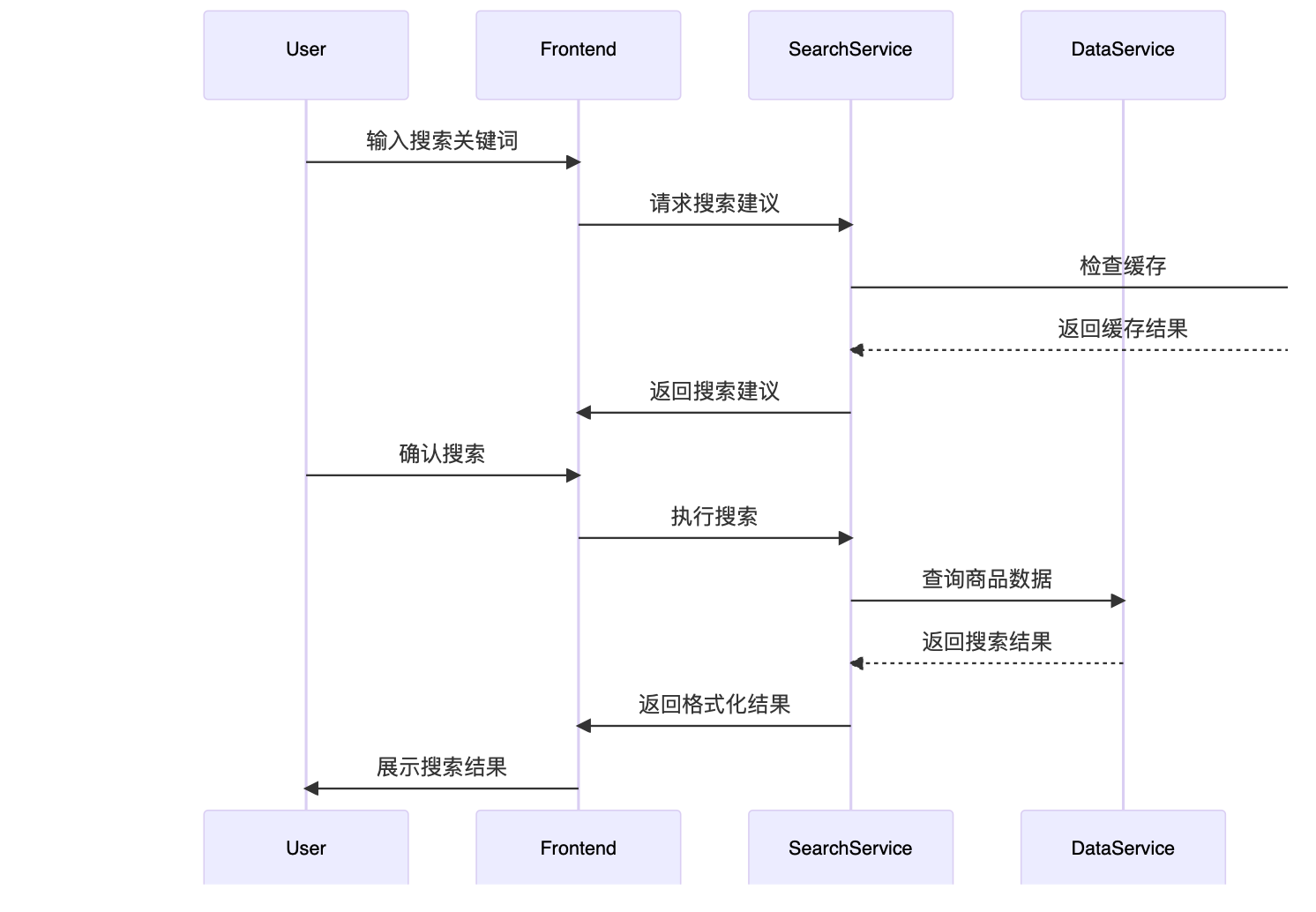
1. 商品名称搜索
   * 支持中文分词搜索
   * 搜索结果排序（价格、平台等）

#### 3.2.2 核心组件设计

1. 搜索栏组件
2. 商品列表组件
3. 筛选组件
4. 状态管理 (SearchStore)

#### 3.2.3 功能实现流程

1. 搜索流程



1. 爬虫流程

* 京东：京东的登录通常需要模拟用户输入用户名、密码以及验证码。可以手动登录京东，获取登录后 cookie，然后将 cookie 传递给爬虫使用。**使用程序自动登录**（例如使用 requests 或 selenium）：模拟登录过程，自动获取 cookie。但自动登录涉及处理验证码，这需要复杂的机制，可以用 selenium 实现模拟登录。根据用户提供的关键词，构建搜索 URL 并请求搜索页面。京东的搜索链接通常包含 keyword（关键词）和 page（页数）参数。使用 BeautifulSoup 解析获取的 HTML 内容，以便从中提取需要的商品信息。在解析后的页面中，定位到包含商品信息的标签，例如商品的 sku、名称、价格 等。通过选择器来定位到包含商品数据的 HTML 元素。 京东的商品信息通常位于 #J\_goodsList > ul 标签下，每个商品的信息会在 li.gl-item 标签中。通过 soup.select() 或 soup.find\_all() 方法，定位到所有商品元素。
* 亚马逊：方法同京东，通过cookie和aiohttp实现爬取
* 购物党：方法同京东，通过cookie和aiohttp实现爬取

### 3.3 价格比较模块





#### 3.3.1 功能描述

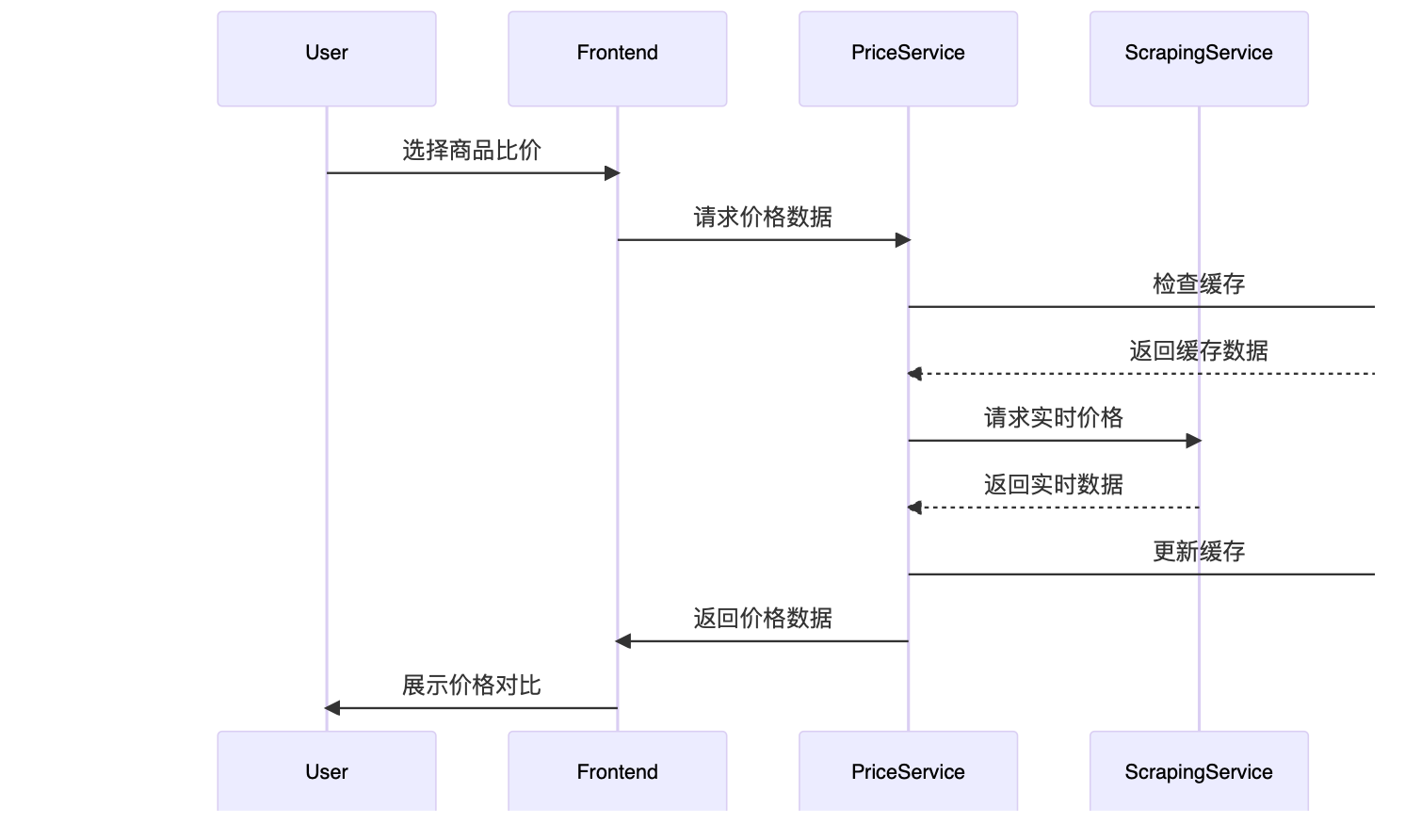
1. 多平台价格展示：同款商品多平台比对，价格、优惠、运费展示，商品详情信息对比，店铺信息与评分展示
2. 价格历史趋势：价格走势图表展示，历史最高/最低价标注，价格波动分析，自定义时间区间查看

#### 3.3.2 核心组件设计

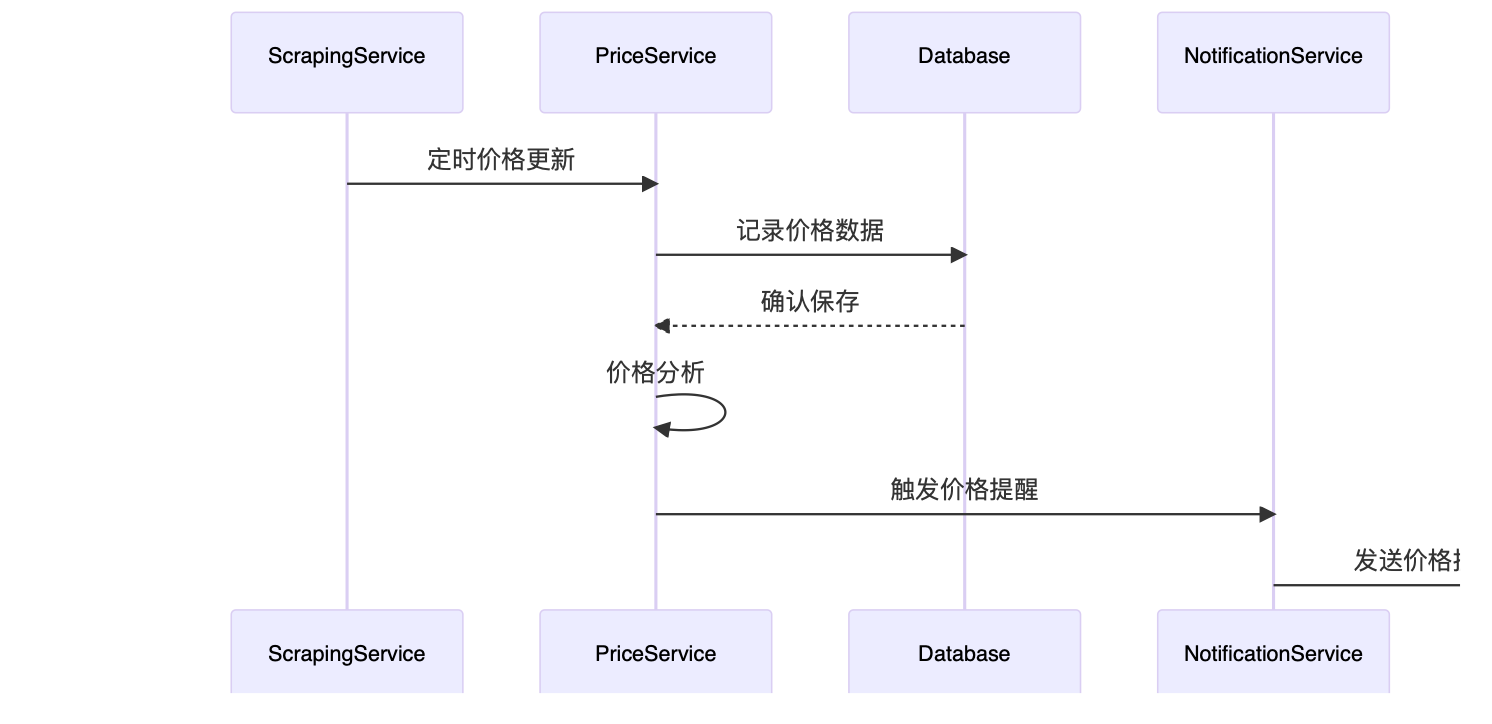
1. 商品详情页面
2. 价格趋势图表
3. 平台列表组件
4. 价格状态管理

#### 3.3.3 功能实现流程

1. 价格比较流程



1. 价格监控更新流程



### 3.4 价格监控模块



#### 3.4.1 功能描述

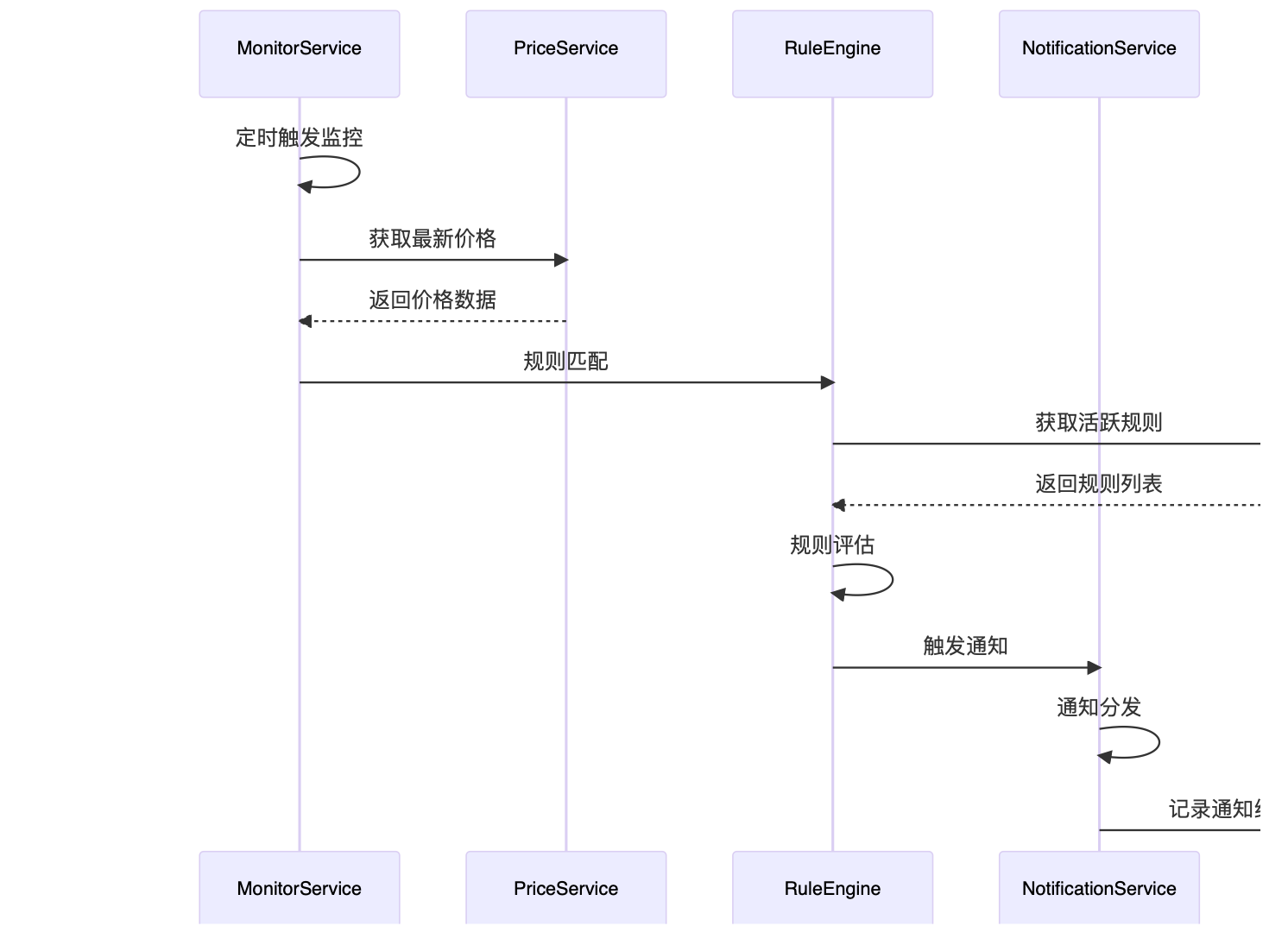
1. 降价提醒设置：目标价格设置、监控时间设置、多商品批量监控
2. 消息推送配置：邮件通知、短信推送、App推送
3. 监控历史记录：价格变动记录、触发记录查看、历史记录导出、数据可视化

#### 3.4.2 核心组件设计

1. 价格提醒设置
2. 提醒列表组件
3. 历史记录组件
4. 监控状态管理 (AlertStore)

#### 3.4.3 功能实现流程

1. 监控任务处理流程

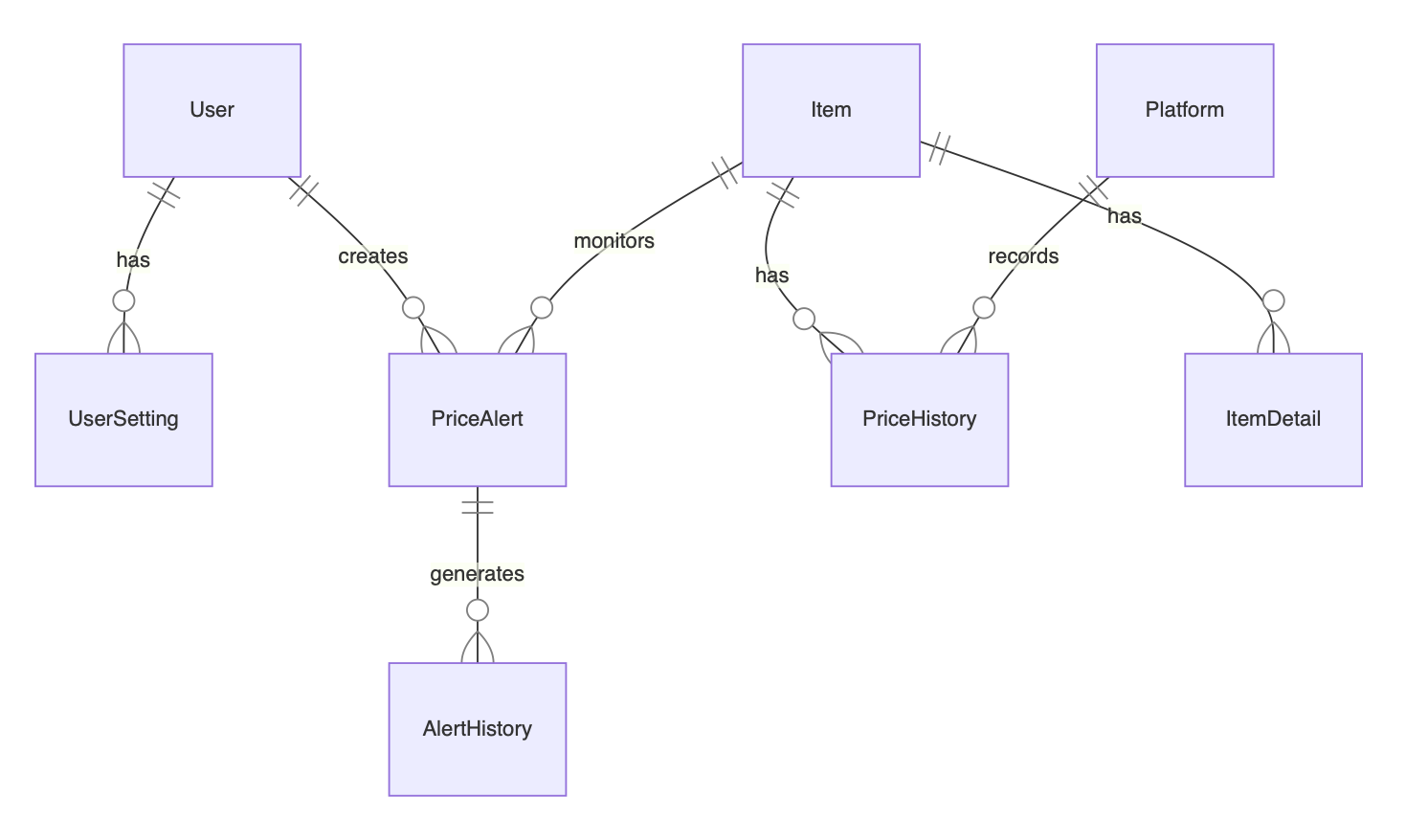


1. 通知发送流程



## 4 数据库设计

### 4.1 数据库架构



### 4.2 核心表结构

1. Item

| 列名 | 数据类型 | 描述 |
| --- | --- | --- |
| id | Integer | 主键 |
| title | String(200) | 商品标题（必填） |
| search\_title | String(255) | 用于中文分词的搜索标题（必填） |
| link | String(255) | 商品链接（必填） |
| image\_url | String(255) | 图片链接（可选） |
| create\_time | DateTime | 创建时间 |
| update\_time | DateTime | 更新时间 |
| current\_price | Float | 当前价格（必填） |
| platform | String(100) | 商品所在的平台（可选） |
| platform\_info | Relationship | 与 Platform 表的关系 |
| shop | String(255) | 商店名称（可选） |
| shop\_link | String(255) | 商店链接（可选） |
| sku | String(255) | SKU 编号（唯一） |
| has\_detail | Boolean | 是否收集过详情 |

**唯一约束：**

* (title, shop) 的联合唯一约束

1. Platform

| 列名 | 数据类型 | 描述 |
| --- | --- | --- |
| id | BigInteger | 主键 |
| name | String(50) | 平台名称（必填，唯一） |
| logo\_url | String(255) | 平台 logo 的链接（可选） |

1. PriceAlert

| 列名 | 数据类型 | 描述 |
| --- | --- | --- |
| id | Integer | 主键 |
| user\_id | Integer | 外键，指向 users 表的用户 ID（必填） |
| item\_id | Integer | 外键，指向 items 表的商品 ID（必填） |
| target\_price | Float | 触发警报的目标价格（必填） |
| created\_at | DateTime | 创建时间 |
| updated\_at | DateTime | 最后更新时间 |
| enable | Boolean | 是否启用该价格提醒 |
| check\_interval | Integer | 检查价格的间隔时间（单位：分钟） |
| notification\_method | String(50) | 提醒通知方式（必填） |

**唯一约束：**

* (user\_id, item\_id) 的联合唯一约束

1. AlertHistory

| 列名 | 数据类型 | 描述 |
| --- | --- | --- |
| id | Integer | 主键 |
| alert\_id | Integer | 外键，指向 PriceAlert 表 |
| price\_before | Numeric(10, 2) | 提醒触发前的商品价格 |
| price\_after | Numeric(10, 2) | 提醒触发后的商品价格 |
| notification\_status | JSON | 通知状态（以 JSON 格式存储） |
| created\_at | DateTime | 创建时间 |

**索引：**

* alert\_id 列的索引
* created\_at 列的索引

1. PriceHistory

| 列名 | 数据类型 | 描述 |
| --- | --- | --- |
| id | Integer | 主键 |
| item\_id | Integer | 外键，指向 items 表的商品 ID（必填） |
| platform\_id | Integer | 外键，指向 platforms 表的平台 ID（必填） |
| price | Numeric(10, 2) | 商品在某一时刻的价格 |
| date | TIMESTAMP | 记录价格的时间 |

**索引：**

* (item\_id, platform\_id) 的联合索引

1. ItemDetail

| 列名 | 数据类型 | 描述 |
| --- | --- | --- |
| id | Integer | 主键 |
| item\_id | Integer | 外键，指向 items 表的商品 ID（必填） |
| platform\_id | Integer | 外键，指向 platforms 表的平台 ID（必填） |
| region | String | 产地 |
| update\_at | TIMESTAMP | 更新的时间 |

## 5 API接口设计

### 5.1 接口规范

#### 5.1.1 基础规范

* 基础路径: /api
* 请求方法: 严格遵循 RESTful 规范
* 响应格式: 统一使用 JSON 格式
* 状态码: 遵循 HTTP 标准状态码
* 认证方式: JWT Token (在 Header 中使用 Authorization: Bearer <token>)

#### 5.1.2 响应格式

{  
 "code": 0, // 业务状态码，0表示成功  
 "message": "success", // 状态描述  
 "data": { // 业务数据  
 // 具体数据结构  
 },  
 "timestamp": "2024-11-13T10:00:00Z" // 响应时间戳  
}

#### 5.1.3 错误处理

{  
 "code": 1, // 错误码  
 "message": "Invalid parameter", // 错误描述  
 "details": { // 详细错误信息  
 "field": "username",  
 "reason": "length must be at least 6 characters"  
 },  
 "timestamp": "2024-11-13T10:00:00Z"  
}

#### 5.1.4 分页参数

* page: 当前页码，默认1
* size: 每页大小，默认20
* totalsize: 总共的结果数量
* pagecount：页面总数

### 5.2 接口清单

#### 5.2.1 用户认证类接口

* 用户注册：/user/signup 路由处理用户注册请求，检查用户名、密码和邮箱是否存在，并将用户信息存储在数据库中。
* 用户登录：/user/login 路由处理用户登录请求，验证用户名和密码，并生成JWT令牌。
* 更新用户信息：/user/auth/loginInfo 路由处理用户信息更新请求，使用JWT令牌验证用户身份，并更新用户信息。

1. 用户注册

POST /api/user/signup  
Request:  
{  
 "username": string, // 用户名  
 "password": string, // 密码  
 "email": string, // 邮箱  
}  
Response:  
{  
 // 通过状态码反映是否注册成功  
}

1. 用户登录

POST /api/user/login  
Request:  
{  
 "username": string, // 用户名/邮箱  
 "password": string, // 密码  
}  
Response:  
{  
 "token": string,  
}

1. 用户登出

GET /api/user/logout  
Request:  
{  
 "username": string, // 用户名/邮箱  
 "password": string, // 密码  
}  
Response:  
{  
 ""  
}

#### 5.2.2 用户信息类接口

1. 获取用户信息 @token\_required

GET /api/user/auth/loginInfo  
Response:  
{  
 "username": string,  
 "email": string,  
 "avatar": string,  
 "watchList": [{  
 "item\_id": integer,  
 "create\_at": Date,  
 "target\_price" float,  
 ...  
 },]  
}

1. 更新用户信息 @token\_required

PUT /api/user/auth/loginInfo  
Request:  
{  
 "username'": string,  
 "email": string,  
 "phone": string  
}  
Response:  
{  
 ""  
}

#### 5.2.3 商品搜索类接口

* 搜索商品：/search 路由处理商品搜索请求，支持分页和排序，并异步抓取更多数据。
* 获取商品详情：/item/<id> 路由处理商品详情请求，返回商品的详细信息，同时使用celery异步更新商品的价格信息。

1. 商品搜索

GET /api/search  
Query:  
- keyword: string // 搜索关键词  
- order: string // 结果排序  
- price\_min: number // 最低价  
- price\_max: number // 最高价  
- platform: string[] // 平台筛选  
- page: number // 页码  
- size: number // 每页数量  
Response: {  
 "total": integer,  
 "pageSize": integer,  
 "pageNo": integer,  
 "totalPages": integer,  
 "items": string  
}  
 serialized\_items: {  
 'id': string  
 'defaultImg': string,  
 'currentPrice': string,  
 'title': string,  
 'link': string,  
 'shopName': string,  
 'shopLink': string,  
 'platform': string  
 }

1. 商品详情

GET /api/items/:id  
Response:  
{  
 "id": string,  
 "title": string,  
 "link": string,  
 "imageUrl": string,  
 "currentPrice": number,  
 "shop": string,  
 "shopLink": string,  
 "platform": string,  
 "attrs": dict,  
 "imgList": list  
}

#### 5.2.4 价格比较类接口

* 获取商品价格历史：/item/price/<id> 路由处理商品价格历史请求，返回商品的价格变化记录。

1. 获取历史价格

GET /api/items/price/:id  
Response:{  
 "price\_history": list[  
 dict{  
 'price':number,   
 'date':date  
 }  
 ...]  
}

#### 5.2.5 价格监控类接口

* 添加价格提醒：/alert 路由处理添加价格提醒请求，检查是否已存在相同的提醒，并将提醒信息存储在数据库中。
* 查询用户提醒：/alert/<uid> 路由处理查询用户提醒请求，返回用户的所有价格提醒。
* 更新价格提醒：/alert/<alertid> 路由处理更新价格提醒请求，修改提醒的目标价格和通知方式。
* 删除价格提醒：/alert/<alert\_id> 路由处理删除价格提醒请求，删除指定的价格提醒。
* 发送通知：/alert/sendemail 和 /alert/sendsms 路由处理发送邮件和短信通知请求。

1. 创建价格提醒

POST /api/alert  
Request:  
{  
 'userId': string,  
 "itemId": string,  
 "targetPrice": number,  
 "notificationMethods": string,  
}

1. 获取提醒列表

GET /api/alert/:uid  
Response:  
{  
 "data": list[  
 {  
 "id": string,  
 "itemId": string,  
 "title": string,  
 "imageUrl": string,  
 "currentPrice": number,  
 "enable": boolean,  
 "targetPrice": number,  
 "notificationMethod": string,  
 "createAt": string,  
 "alertHistory" : histories[],  
 "priceHistory" : price\_hist[]  
 }  
 ]  
}

1. 更新提醒设置

PUT /api/alert/:alertid  
Request:  
{  
 "targetPrice": number,  
 "notificationMethods": string[],  
 "enable": boolean,  
}

1. 获取提醒历史

GET /api/alert/history/:alertid  
Response:  
{  
 "data": list[{  
 "id": history\_id,  
 "priceBefore": number,  
 "priceAfter": number,  
 "notificationStatus": boolean,  
 "createAt": string,  
 }]  
}

1. 删除提醒

DELETE /api/:alertid  
Response:  
{  
 "success"  
}

### 5.3 接口权限控制

#### 5.3.1 权限等级

1. 公开接口（无需认证）

* 商品搜索
* 商品分类
* 价格对比查看

1. 用户接口（需要认证）

* 用户信息操作
* 搜索历史
* 价格监控设置

## 6 项目结构设计

### 6.1 项目目录结构

#### 6.1.1 后端结构

back-end  
├─ Dockerfile  
├─ README.md  
├─ app  
│ ├─ \_\_init\_\_.py  
│ ├─ celeryconfig.py  
│ └─ config.py  
├─ celery\_route  
│ ├─ \_\_init\_\_.py  
│ └─ test.py  
├─ celerybeat-schedule.db  
├─ controllers  
│ ├─ \_\_init\_\_.py  
│ ├─ alert\_controller.py  
│ ├─ item\_controller.py  
│ ├─ platform\_controller.py  
│ ├─ price\_controller.py  
│ └─ user\_controller.py  
├─ data  
│ ├─ Amazoncookie.txt  
│ ├─ GWcookie.txt  
│ └─ JDcookie.txt  
├─ models  
│ ├─ \_\_init\_\_.py  
│ ├─ item.py  
│ ├─ platform.py  
│ ├─ price\_alert.py  
│ ├─ price\_history.py  
│ ├─ search\_history.py  
│ └─ user.py  
├─ requirements.txt  
├─ routes  
│ ├─ \_\_init\_\_.py  
│ ├─ alert.py  
│ ├─ search.py  
│ └─ user.py  
├─ run.py  
└─ utils  
 ├─ \_\_init\_\_.py  
 ├─ auth.py  
 ├─ crawler.py  
 └─ tasks.py

1. app内存放了初始化flask、celery、mysql等功能的代码，config文件配置了网络参数
2. data内存放了各个电商网站的cookie
3. controllers主要负责实现和各个数据库model的交互
4. models内定义了各个table的数据
5. routes内定义了后端如何接收前端的请求以及各个api的情况
6. utils中实现了爬虫功能、celery的异步检查价格功能和发送短信邮件功能

#### 6.1.2 前端结构

front-end  
├─ Dockerfile  
├─ README.md  
├─ auto-imports.d.ts  
├─ components.d.ts  
├─ default.conf  
├─ index.html  
├─ package.json  
├─ public  
│ └─ favicon.ico  
├─ src  
│ ├─ App.vue  
│ ├─ api  
│ │ ├─ index.ts  
│ │ └─ request.ts  
│ ├─ assets  
│ │ └─ images  
│ ├─ components  
│ │ ├─ PriceChart  
│ │ │ └─ index.vue  
│ │ ├─ footer  
│ │ │ └─ Footer.vue  
│ │ ├─ header  
│ │ │ └─ index.vue  
│ │ └─ message  
│ │ └─ index.vue  
│ ├─ config  
│ ├─ env.d.ts  
│ ├─ libs  
│ │ ├─ token.ts  
│ │ └─ utils.ts  
│ ├─ main.ts  
│ ├─ mock  
│ │ ├─ home  
│ │ │ ├─ banner.json  
│ │ │ └─ floor.json  
│ │ ├─ home.ts  
│ │ └─ user.ts  
│ ├─ router  
│ │ └─ index.ts  
│ ├─ store  
│ │ ├─ alerts.ts  
│ │ ├─ item.ts  
│ │ ├─ login.ts  
│ │ ├─ search.ts  
│ │ └─ user.ts  
│ ├─ styles  
│ ├─ types  
│ │ ├─ index.ts  
│ │ ├─ login  
│ │ │ └─ index.ts  
│ │ ├─ search  
│ │ │ └─ index.ts  
│ │ └─ user  
│ │ └─ index.ts  
│ └─ views  
│ ├─ home  
│ │ ├─ index.vue  
│ │ └─ search-bar  
│ │ └─ index.vue  
│ ├─ item  
│ │ ├─ PriceChart  
│ │ │ └─ index.vue  
│ │ └─ index.vue  
│ ├─ price-alert  
│ │ ├─ PriceChart  
│ │ │ └─ index.vue  
│ │ ├─ UserPanel  
│ │ │ └─ index.vue  
│ │ └─ index.vue  
│ ├─ register  
│ │ └─ index.vue  
│ └─ search  
│ └─ index.vue  
├─ tsconfig.json  
└─ vite.config.ts

1. Components：采用组件化设计，将页面拆分为多个独立的组件，便于维护和复用。Header、Footer、PriceChart等组件分别实现各自的功能。
2. Store：使用 Pinia 进行状态管理，将全局状态存储在 store 目录下的各个模块中。例如，login.ts 管理登录状态，search.ts 管理搜索状态，alerts.ts 管理价格提醒状态。
3. Router：用 Vue Router 进行路由管理，将不同的页面和组件通过路由进行连接。路由配置文件位于 [index.ts](vscode-file://vscode-app/Applications/Visual%20Studio%20Code.app/Contents/Resources/app/out/vs/code/electron-sandbox/workbench/workbench.html)，定义了各个页面的路由规则。
4. api：将所有的 API 请求封装在api目录下，通过request.ts进行统一管理。使用 Axios 进行 HTTP 请求，并在请求和响应拦截器中处理通用逻辑。
5. View：借助vue框架实现了各个页面的内容以及函数逻辑

## 7 Docker部署方案

Docker利用容器来运行前端、后端和数据库服务，每个服务都在独立的容器中运行，互相隔离又可通过网络互联。

Docker分别为前端、后端和数据库创建独立的容器。例如，前端的Vue应用、后端的Flask API和MySQL数据库都在独立的容器中运行，相当于在同一台主机上拥有多个轻量级的“沙盒”。Docker提供了一个内部网络，容器之间可以通过Docker网络中的服务名称相互访问。

所以需要编写Dockerfile构建前后端镜像，配置Docker Compose来协调前后端和数据库的启动，最后进行容器化的访问。

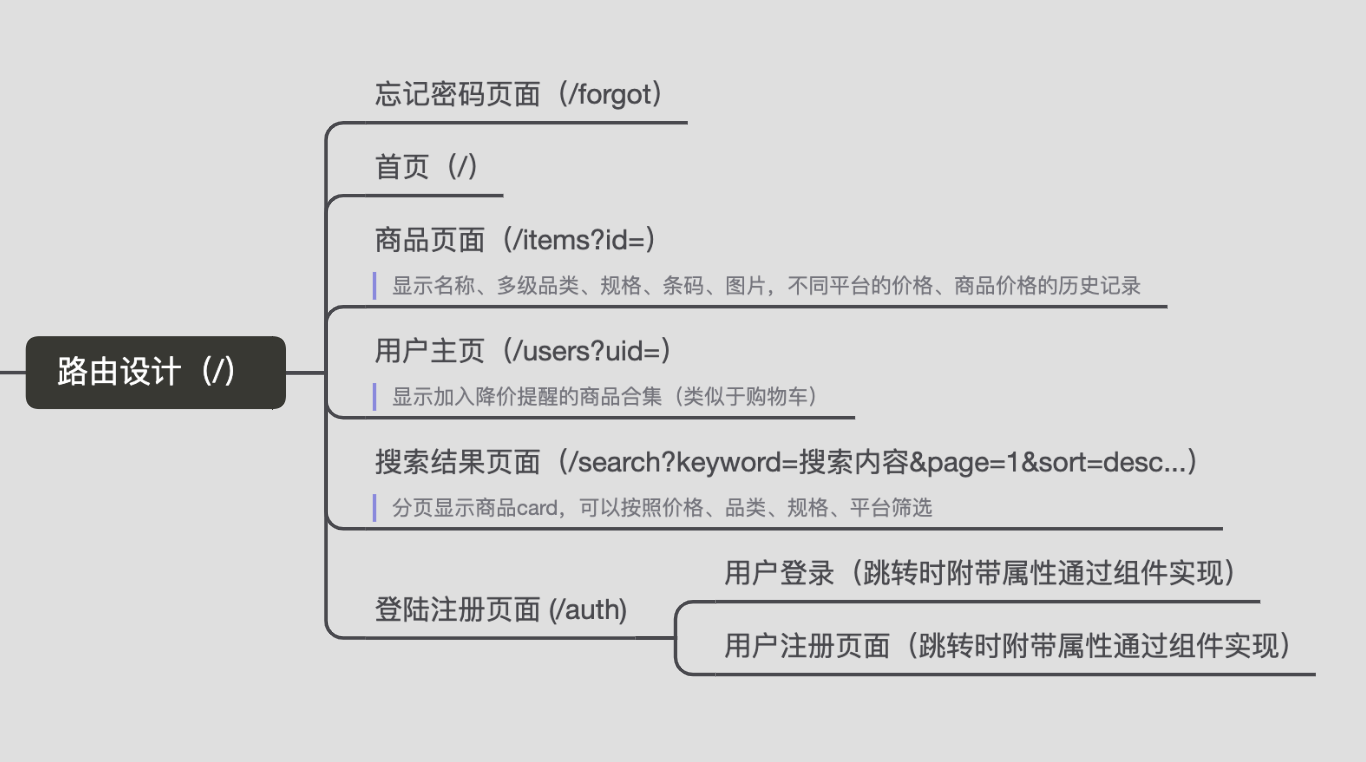
### 7.1 部署配置文件

version: '3.9'  
  
services:  
 flask:  
 build:  
 context: ./back-end  
 image: zhouyueer/price-comparison-system-flask:latest  
 container\_name: flask\_app  
 ports:  
 - "5001:5000"  
 depends\_on:  
 - redis  
 - mysql  
 restart: always  
  
 vue:  
 build:  
 context: ./front-end  
 image: zhouyueer/price-comparison-system-vue:latest  
 container\_name: vue\_app  
 depends\_on:  
 - flask  
 restart: always  
  
 nginx:  
 build:  
 context: ./front-end  
 image: zhouyueer/price-comparison-system-nginx:latest  
 container\_name: nginx  
 ports:  
 - "80:80"  
 depends\_on:  
 - vue  
 - flask  
 restart: always  
  
 mysql:  
 image: mysql:8.0  
 container\_name: mysql\_db  
 restart: always  
 environment:  
 MYSQL\_ROOT\_PASSWORD: rootpassword  
 MYSQL\_DATABASE: pricecomp  
 MYSQL\_USER: user  
 MYSQL\_PASSWORD: password  
 ports:  
 - "3307:3306"  
 volumes:  
 - mysql\_data:/var/lib/mysql  
 - ./sql:/docker-entrypoint-initdb.d  
  
 redis:  
 image: redis:latest  
 container\_name: redis  
 restart: always  
 ports:  
 - "6379:6379"  
  
 celery:  
 build:  
 context: ./back-end  
 image: zhouyueer/price-comparison-system-celery:latest  
 container\_name: celery\_worker  
 command: celery -A run.celery worker --loglevel=info  
 depends\_on:  
 - flask  
 - redis  
 restart: always  
  
 celery-beat:  
 build:  
 context: ./back-end  
 image: zhouyueer/price-comparison-system-celery-beat:latest  
 container\_name: celery\_beat  
 command: celery -A run.celery beat --loglevel=info  
 depends\_on:  
 - flask  
 - redis  
 restart: always  
  
volumes:  
 mysql\_data:

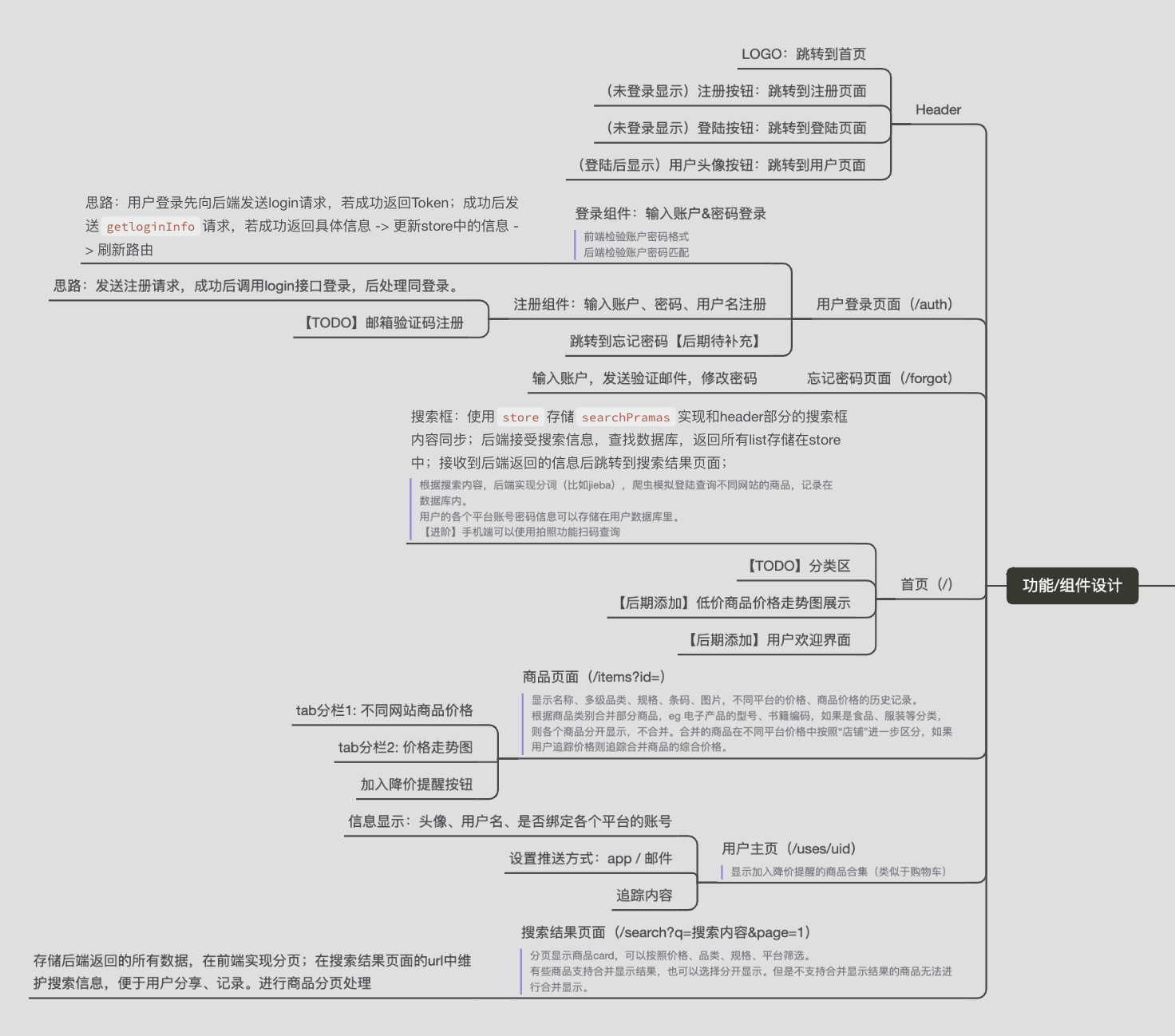
* **前端访问**：通过http://localhost访问前端应用，Vue应用会静态打包后由Nginx服务提供。
* **后端访问**：通过http://localhost:5001访问后端API。
* **数据库连接**：后端Flask应用可以通过db:3306访问数据库服务
* 启动后，前端可以通过主机80端口访问，后端通过5001端口访问，内部服务之间通过nginx反向代理相互通信，实现高效、稳定的部署方案。

## 8 附录

### 8.1 路由设计思维导图



### 8.2 组件设计思维导图



### 8.3 LOGO设计



