基于 Angular 和 NestJS 的动漫平台评论网站系统

佘嘉洛*

计算机科学与技术学院软件工程专业

Made By LaTeX

2021 年 12 月 25 日

*电子邮件: shejialuo@stu.xidian.edu.cn, 学号：21031211515

1. 项目简介

本项目拟实现一个全栈的动漫平台评论网站系统。实现如下的功能：

- 用户注册功能
  - 前端表单验证用户两次输入的密码是否一致。
  - 后端验证用户注册的用户名是否重名，前端需防抖。
  - 发送邮件验证，用户需输入正确的验证码才能注册。
- 用户登录功能
  - 前端用户为登录时，用户使用用户名和密码登陆
  - 第一次登录，后端验证利用 JWT 生成 token，前端通过 Response 对象获取 token，存储在 LocalStorage 中，不使用 cookie，避免 CSRF。
  - 登录后，在 token 有效期内，前端用户无需再次登陆。
- 用户评论功能

2. 技术栈选取

本项目采用前后端分离开发的方式进行。

2.1 前端

本项目前端采取 Angular 框架进行开发。Angular 框架是由是一个基于 TypeScript 的开源 Web 应用框架由 Google 的 Angular 团队以及社区共同领导。同时，本项目采用 Angular 官方的基于 Material Design 的样式库 Angular Material 作为组件开发库：

- Angular Version: 13.0.3
- Angular Material Version: 13.0.1

2.2 数据库

本项目使用非关系型数据库 MongoDB 作为数据库，采用 Mongoose 作为数据库处理高层语言。

2.3 后端

本项目使用 NestJS 作为后端开发。

3. 项目架构设计

本节阐述本项目的架构设计，整体架构设计如图 1 所示。

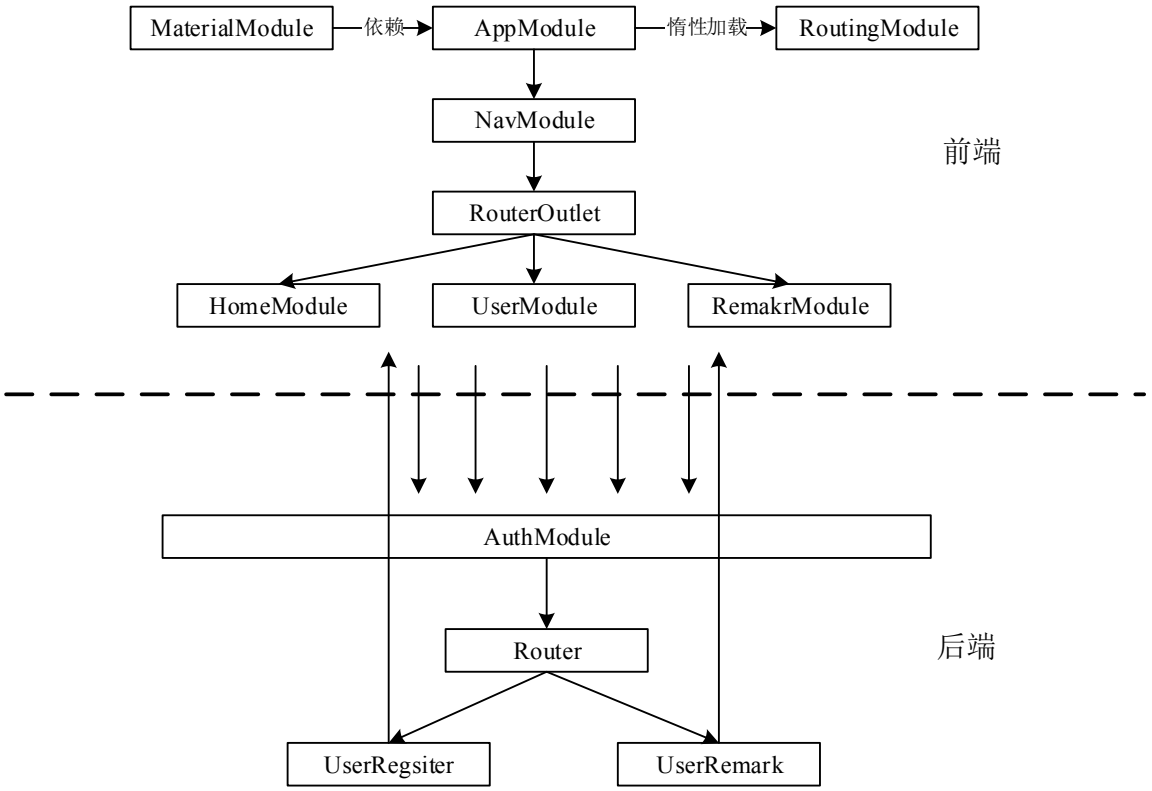

图 1: 项目整体架构示意图

4. 关键技术解决

本节阐述在开发过程中解决的关键技术问题

4.1 数据库设计

由于采用非关系性数据库且本项目业务逻辑较简单，本项目的数据库设计比较简单，本项目设置了两个类`User`, `Remark` 即完成了数据库设计，如下所示：

```
@Schema()
export class User {
  @Prop({ required: true })
  userName: string;

```

```
    @Prop({ required: true })
    userPassword: string;

    @Prop({ required: true })
    userEmail: string;
  }
  @Schema()
  export class Remark {
    @Prop({ required: true })
    userName: string;

    @Prop({ required: true })
    userRemark: string;
  }
}

```

Listing 1: 数据库设计

4.2 注册表单实现

表单的实现是一个关键，主要实现的还是验证的功能。前端的表单验证分为了两个类型：

- 前端直接验证
- 后端验证

本项目对于这两种类型的验证均涉及到了。首先是前端验证用户输入的密码以及确认密码是否一致，本项目通过 Angular 框架自带的响应式表单很容易地实现了前端直接验证。整个表单通过`FormGroup`控制，表单的字段通过`FormControl`控制。通过响应式表单，由函数控制表单，而不是在`HTML`模板中使用模板变量控制，从而实现了模板与控制器的解耦。

4.2.1 用户名重名验证

由图 2 可以看出，对于第一个步骤“请填写您的相关信息”，本文通过建立一个关键的`FormGroup`类`passwordFormGroup`控制，同时建立三个`FormControl`，如下所示：

```
    this.passwordFormGroup = this.formBuilder.group({
```

图 2: 前端表单步进器设计

```
2       userName: ['', [Validators.required, Validators.minLength
    (6)], this.userNameValidators],
3       userPrePassword: ['', [Validators.required, Validators.
    minLength(6)]],
4       userPostPassword: ['', [Validators.required, this.
    passwordValidators]],
5     });
6
```

Listing 2: 响应式表单代码

可以看出，对于注册的用户名，本项目使用内置的同步验证器表明用户必须输入，且至少需要 6 位。同时，本项目定义了一个自定义的异步验证器用于查询用户名是否重名。显然，此步需要进行防抖的操作。

```
 userNameValidators: AsyncValidatorFn = (control:
AbstractControl) => {
   return control.valueChanges.pipe(
     debounceTime(1000),
     distinctUntilChanged(),
     switchMap(() => {
       return this.userService.checkName(control.value);
     }),
     map(result => result ? {isSame: true}: null),
     first(),
   )
 }

```

Listing 3: 验证用户名是否重名

从上述代码可以看出，每当`FormControl`的值改变时，将会进行一系列的判断。首先时用户在 1s 内没有执行输入的操作，且输入的值没有发生改变，往后端发送请求，查询数据库是否存在同名的用户名。

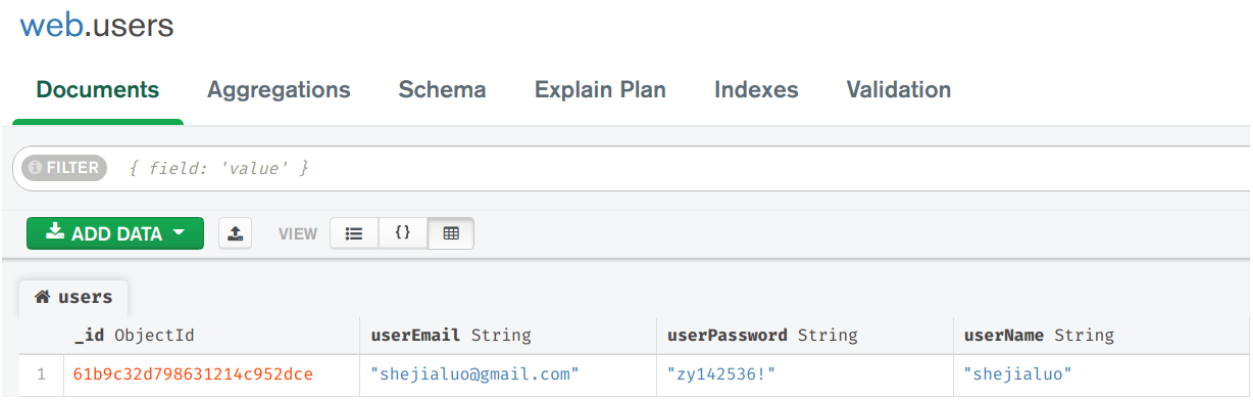

图 3: MongoDB 数据库展示

如图 3 所示，此时系统已经注册了一位用户名为`shejialuo`的用户。此时，如图 4 所示，在注册表单输入用户名`shejialuo`将会提示红字用户名已经重名请修改。

4.2.2 验证输入密码一致

4.2.1 节中阐述的是异步验证，也就是通过后端请求的数据返回前端做表单验证，本节将阐述同步验证。其实现的思路比较简单，就是比较值，此处不赘述，如图 5 所示。

图 4: 用户名重名示意图

4.2.3 邮箱验证

邮箱验证实现的思路是比较简单的，对于前端而言，首先要对邮件的格式做验证。后发送请求至后端。后端通过产生 4 位数的随机码，通过 SMTP 发送即可。为了维持状态，维持一个`MAP<emailAddress, checkCode>`即可。

4.3 登录

本小节阐述本项目如何实现登录功能，本项目通过 JWT 实现验证。其基本的思想是用户通过输入用户名和密码登录，后端进行验证，通过 JWT 返回 token。前端接收到后端的 Response，提取出 token，将其保存到 LocalStorage 中，后用户在有效期内，可以直接免密登录。

图 5: 用户输入密码不一致示意图

4.3.1 后端 JWT 生成

由于 NestJS 框架提供了内置的 JWT 模块，可以直接通过依赖注入实现功能。故对于用户使用用户名和密码登录的情况，本项目查询数据库，如果满足，则发送 JWT Token 提供给前端。

```
1     @Post('login')
2     async login(@Res({passthrough: true})
3           response: FastifyReply ,
4           @Body() loginDto: LoginDto) {
5       let loginSuccess = await this.authService.userLogin(
    loginDto);
6
7       if(!loginSuccess) {
```

```
        response.send({success: false});
      }
      else {
        const userId = loginDto.userName;
        const payload = {userId: userId};
        const token = this.jwtService.sign(payload);
        response.send({
          success: true,
          jwtToken: token});
      }
    }

```

Listing 4: 后端 JWT 生成并传递给前端

4.3.2 前端保存 Token

前端需要把后端发送的 Token 保存到 LocalStorage 中，这一点相对容易比较实现。此处就不赘述了。

4.3.3 前端实现 Token 注入 Header

由于本项目将 Token 保存在 LocalStorage 中，不像 cookie 在每次进行 HTTP 通信时，会自动将其注入 Header。因此，在每一次进行 HTTP 通信时，都需要把 Token 注入 Header 中，以便于用于后端的验证。显然，这一步最好透明化，Angular 框架提供了 HTTP Interceptor 机制可以透明地实现这一功能：

```
export class AuthInterceptor implements HttpInterceptor {
intercept(req: HttpRequest<any>,
          next: HttpHandler): Observable<HttpEvent<any>> {

  const idToken = localStorage.getItem("id_token");

  if (idToken) {
    const cloned = req.clone({
    headers: req.headers.set("Authorization",
              idToken)
    });

    return next.handle(cloned);
```

```
14        }
15        else {
16          return next.handle(req);
17        }
18      }
19  }
20
```

Listing 5: 前端 Header 自动注入 Token

通过如上的操作既可以透明实现 Token 注入 Header。

5. 总结

麻雀虽小，五脏俱全。