

基于LocalStorage和用户ID传递的Web应用认证机制实现

邓睿涵
杭州云谷学校, 杭州, 中国
Email: raymond.dengruhan@yungu.org

摘要

本文详细阐述了一个WiFi评级Web应用的用户登录与授权实现机制。该应用采用基于LocalStorage的客户端状态管理和用户ID显式传递的认证方式，而非传统的Cookie/Session或JWT Token机制。本文从浏览器端状态管理、服务器端用户识别、密码验证流程等四个核心问题出发，深入分析了认证系统的设计原理与实现细节，并通过代码示例和流程图说明了完整的认证流程。

关键词—用户认证, LocalStorage, Django REST Framework, 密码哈希, Web安全

I. 引言

现代Web应用的用户认证机制通常采用Cookie/Session或JWT Token等标准方案。然而，本文所研究的WiFi评级应用采用了一种基于LocalStorage和用户ID显式传递的简化认证机制。该机制虽然不如传统方案完善，但在特定场景下能够满足基本的安全需求。本文旨在详细分析该认证机制的实现原理，回答以下四个核心问题：

1. 浏览器端如何管理授权数据，确保每次请求自动携带用户身份信息？
2. 服务器如何识别用户并生成/传递认证凭证？
3. 服务器如何存储和管理认证凭证以关联用户身份？
4. 服务器如何验证用户名和密码的正确性？

II. 系统架构概述

本应用采用前后端分离架构：

- 前端：Vue.js 3 + Vite，运行在 <http://localhost:5173>
- 后端：Django 6.0 + Django REST Framework，运行在 <http://localhost:8000>
- 数据库：SQLite（开发环境）

认证机制的核心特点：

- 使用浏览器LocalStorage存储用户信息（JSON格式）
- 前端在每次API请求中显式传递用户ID
- 后端通过用户ID直接识别用户，无需Token验证
- 使用Django内置的密码哈希机制进行密码验证

III. 问题一：浏览器端授权数据管理机制

A. LocalStorage存储机制

本应用未使用Cookie或Token，而是采用浏览器LocalStorage存储用户信息。登录成功后，用户数据以JSON格式存储在LocalStorage中：

```
// 登录成功后保存用户信息
localStorage.setItem('user', JSON.stringify(data.user))
```

存储的用户信息包括：

```
{
  "id": 1,
  "username": "张三",
  "email": "zhangsan@example.com",
  "date_joined": "2024-01-01T00:00:00Z",
  "avatar": "data:image/png;base64,..."}
```

B. 应用启动时的状态恢复

应用启动时（App.vue的`onMounted`钩子），会从LocalStorage读取用户信息并恢复登录状态：

```
onMounted(() => {
  const savedUser = localStorage.getItem('user')
  if (savedUser) {
    try {
      const user = JSON.parse(savedUser)
      isloggedIn.value = true
      currentUser.value = user
    } catch (error) {
      localStorage.removeItem('user')
    }
  }
})
```

C. 请求中的用户ID传递

由于未使用Cookie自动传递机制，本应用需要在每个需要用户身份的API请求中显式传递用户ID。前端通过Vue的依赖注入（inject）机制获取当前用户ID，并在请求体中包含：

示例1：提交评价请求

```
await axios.post('http://127.0.0.1:8000/api/reviews/', {
  userId: this.currentUser.value.id,
  wifiModelId: id,
  rating: this.newReview.rating,
  comment: this.newReview.comment,
  isAnonymous: false
})
```

示例2：收藏操作请求

```
await axios.post('http://127.0.0.1:8000/api/favorites/', {
  userId: this.currentUser.id,
  wifiModelId: wifiId
})
```

示例3：获取用户收藏列表

```
axios.get(`http://127.0.0.1:8000/api/favorites/${this.currentUser.value.id}/`)
```

D. 机制总结

本应用的授权数据传递机制特点：

- 优点：实现简单，无需配置Cookie或Token管理
- 缺点：每次请求需手动传递用户ID，容易遗漏；安全性较低，用户ID暴露在请求体中

V. 问题三：服务器端Token/Session管理

A. 无服务器端Session管理

本应用未实现服务器端的Session或Token存储机制。Django REST然启用了SessionAuthentication，但实际未使用：

```
# settings.py
REST_FRAMEWORK = {
  'DEFAULT_PERMISSION_CLASSES': [
    'rest_framework.permissions.AllowAny',
  ],
  'DEFAULT_AUTHENTICATION_CLASSES': [
    'rest_framework.authentication.SessionAuthentication',
  ]
}
```

由于所有API接口都设置为AllowAny权限，服务器不验证请求的信任前端传递的用户ID。

B. 用户身份识别方式

服务器通过以下方式识别用户：

1. 从请求体获取用户ID（POST/PUT/PATCH请求）：

```
user_id = request.data.get('userId')
user = User.objects.get(id=user_id)
```

2. 从URL路径获取用户ID（GET请求）：

```
# URL: /api/favorites/1/
favorites = Favorite.objects.filter(user_id=user_id)
```

C. 安全风险分析

当前实现的安全隐患：

- 无Token过期机制
- 无Token撤销机制
- 用户ID直接暴露在请求中，容易被篡改
- 服务器不验证请求的真实性

改进建议：

1. 实现JWT Token机制
2. 添加Token过期时间
3. 服务器端验证Token有效性
4. 使用HTTPS加密传输

VI. 问题四：密码验证机制

A. Django密码哈希机制

Django使用PBKDF2（Password-Based Key Derivation Function 2）处理。用户注册时，密码自动被哈希并存储：

```
# api/serializers.py - UserSerializer
def create(self, validated_data):
  user = User(
    username=validated_data['username'],
    email=validated_data['email']
  )
  user.set_password(validated_data['password'])
  user.save()
  return user
```

set_password()方法内部使用PBKDF2算法，生成格式如下的哈希：

```
pbkdf2_sha256$26000$salt$hashed_password
```

B. 密码验证流程

登录时的密码验证流程：

```
# 步骤1：通过邮箱查找用户
user = User.objects.get(email=email)

# 步骤2：使用Django authenticate()验证
user = authenticate(request, username=user.username, password=password)
```

authenticate()函数的工作流程：

1. 从数据库读取用户的密码哈希值
2. 使用相同的PBKDF2算法和盐值对输入的密码进行哈希
3. 比较哈希结果是否一致
4. 返回用户对象（验证成功）或None（验证失败）

C. 密码安全性保障

Django的密码哈希机制提供以下安全保障：

- 单向哈希：无法从哈希值反推原始密码
- 盐值（Salt）：每个密码使用随机盐值，防止彩虹表攻击
- 迭代次数：PBKDF2默认260000次迭代，增加破解难度
- 自动更新：如果Django升级哈希算法，旧密码会在用户下次登录时自动重加密

IV. 问题二：服务器端用户识别与认证凭证生成

A. 登录流程

完整的登录流程如下：

```
用户输入邮箱和密码
前端发送POST请求到 /api/login/
后端查找用户（通过邮箱）
    ↓
Django authenticate()验证密码
验证成功 → 返回用户信息 (JSON)
前端保存到LocalStorage
```

B. 登录API实现

后端登录接口 (`api/views.py`) :

```
@api_view(['POST'])
def login(request):
    if request.method == 'POST':
        email = request.data.get('email')
        password = request.data.get('password')

        # 步骤1: 查找用户
        try:
            user = User.objects.get(email=email)
        except User.DoesNotExist:
            return Response(
                {'message': '邮箱或密码错误'},
                status=status.HTTP_401_UNAUTHORIZED
            )

        # 步骤2: 验证密码
        user = authenticate(request, username=user.username, password=password)
        if user is not None:
            # 步骤3: 返回用户信息（作为“凭证”）
            return Response(
                {'message': '登录成功！',
                 'user': {
                     'id': user.id,
                     'username': user.username,
                     'email': user.email,
                     'date_joined': user.date_joined,
                     'avatar': user.avatar
                 }},
                status=status.HTTP_200_OK
            )
        else:
            return Response(
                {'message': '邮箱或密码错误'},
                status=status.HTTP_401_UNAUTHORIZED
            )
    
```

VII. 完整认证流程图



VIII. 代码实现细节

A. 前端登录组件 (Login.vue)

```
async handleLogin() {
  const response = await fetch('http://127.0.0.1:8000/api/login/', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({
      email: this.email,
      password: this.password
    })
  })
  if (response.ok) {
    const data = await response.json()
    localStorage.setItem('user', JSON.stringify(data))
    this.$router.push('/home')
  } else {
    alert('登录失败')
  }
}
```

IX. 安全性分析与改进建议

A. 当前实现的安全问题

- 无Token机制：用户ID直接暴露，容易被伪造
- 无过期机制：登录状态永久有效，直到用户主动退出
- 无服务器验证：服务器完全信任前端传递的用户ID
- HTTP传输：开发环境使用HTTP，密码和用户信息明文传输

B. 改进方案

- 方案1：实现JWT Token

```

        })
    })
}

const data = await response.json()
if (response.ok && data.user) {
    this.login(data.user)
    localStorage.setItem('user', JSON.stringify(data.user))
    this.$router.push('/dashboard')
}

```

B. 全局状态管理 (App.vue)

```

const isLoggedIn = ref(false)
const currentUser = ref(null)

const login = (user) => {
    isLoggedIn.value = true
    currentUser.value = user
}

provide('isLoggedIn', isLoggedIn)
provide('currentUser', currentUser)
provide('login', login)

```

C. 后端用户模型 (models.py)

```

from django.contrib.auth.models import AbstractUser

class User(AbstractUser):
    email = models.EmailField(unique=True)
    avatar = models.TextField(blank=True, null=True)

    class Meta:
        db_table = 'users'

```

继承AbstractUser自动获得:

- 密码哈希字段 (password)
- 用户名、邮箱等基础字段
- `set_password()` 和 `check_password()` 方法

参考文献

- [1] Django Software Foundation, "Django Documentation," 2024. [Online]. Available: <https://docs.djangoproject.com/>
- [2] Vue.js Team, "Vue.js Documentation," 2024. [Online]. Available: <https://vuejs.org/>
- [3] Django REST Framework, "DRF Authentication," 2024. [Online]. Available: <https://www.django-rest-framework.org/api-guide/authentication/>

```

# 登录成功后生成JWT
import jwt
token = jwt.encode({
    'user_id': user.id,
    'exp': datetime.utcnow() + timedelta(days=7)
}, SECRET_KEY, algorithm='HS256')

```

方案2: 使用Django Session

```

# 登录时创建Session
request.session['user_id'] = user.id
# 后续请求验证Session
user_id = request.session.get('user_id')

```

方案3: 添加请求签名

```

# 前端生成请求签名
signature = hmac.new(SECRET_KEY, f"{user_id}{timestamp}".encode())
# 后端验证签名

```

X. 结论

本文详细分析了一个基于LocalStorage和用户ID传递的Web应用认证机制。该方案简单，但存在明显的安全缺陷。主要特点包括：

1. 浏览器端: 使用LocalStorage存储用户信息，每次请求显式传递用户ID
2. 服务器端: 直接通过用户ID识别用户，无Token/Session管理
3. 密码验证: 使用Django的PBKDF2哈希机制，安全性较高
4. 安全风险: 缺乏Token验证、过期机制和请求签名

对于生产环境，建议采用JWT Token或Django Session机制，并启用HTTPS加密保证系统的安全性和可靠性。