基于 Django Session 机制的用户登录与授权系统实现

1st 张翼
计算机科学与技术系
杭州云谷学校
杭州, 中国
tyler.zhangyi@yungu.org

Abstract

本文阐述基于 Django Session 机制的用户登录与授权系统实现，回答四个核心问题：Cookie 在浏览器中的工作原理及自动认证机制、服务器如何识别用户并发放 Cookie、Session 存储与用户关联方式、密码验证流程。

Index Terms

用户认证, Session 管理, Cookie 机制, Django 框架, Web 安全

I. 引言

本文基于 Vue.js 前端和 Django 后端的 Web 应用系统，阐述 Django Session 认证机制的实现，回答以下四个问题：(1) Cookie 在浏览器中的工作原理及多请求自动认证；(2) 服务器如何识别用户并发放 Cookie；(3) Session 存储与用户关联机制；(4) 密码验证流程。

II. Cookie 机制与自动认证

A. *Cookie* 在浏览器中的工作原理

HTTP Cookie 存储在浏览器中，服务器通过 Set-Cookie 响应头设置 Cookie 后，浏览器会在后续请求中自动在请求头中包含该 Cookie。前端使用 Axios 库，关键配置如下：

```
import axios from 'axios'

const client = axios.create({
  baseURL: import.meta.env.VITE_API_BASE_URL ||
           'http://localhost:8000/api/',
  withCredentials: true  // 启用 cookie，用于 session 认证
})

export default client
```

Listing 1. 前端 API 客户端配置

`withCredentials: true` 使浏览器在跨域请求中自动包含 Cookie。

B. 多请求场景下的自动认证

登录成功后，服务器通过 Set-Cookie 设置 `sessionid` Cookie。此后，浏览器在每次 AJAX 请求中自动在请求头包含该 Cookie，无需前端代码处理。前端登录实现：

```
async function login(account, password) {
  loading.value = true
  error.value = ''
  try {
    const res = await client.post('auth/login', {
      account, password
    })
    const user = res.data.data
    currentUser.value = { ...user, name: user.account,
                          id: user.user_id }
```

```
    localStorage.setItem('currentUser',
                         JSON.stringify(currentUser.value))
    return { success: true, message: res.data.message }
  } catch (e) {
    error.value = e.message
    return { success: false, message: e.message }
  } finally {
    loading.value = false
  }
}
```

Listing 2. 前端登录逻辑

`localStorage` 仅用于前端状态管理，认证依赖浏览器自动发送的 Cookie。

服务器端 Cookie 配置：

```
SESSION_COOKIE_NAME = "sessionid"
SESSION_COOKIE_AGE = 86400 * 7  # 7天过期
SESSION_COOKIE_HTTPONLY = True
SESSION_COOKIE_SAMESITE = "Lax"
SESSION_COOKIE_SECURE = False  # 开发环境
```

Listing 3. Session Cookie 配置

`HttpOnly=True` 防止 JavaScript 访问，`SameSite=Lax` 限制跨站发送，`Secure` 在生产环境应设为 True。

III. 服务器端用户识别与会话创建

A. 登录流程概述

登录流程如图 1所示。用户提交登录表单后，前端通过 POST 请求将账号和密码发送到`/api/auth/login` 端点。

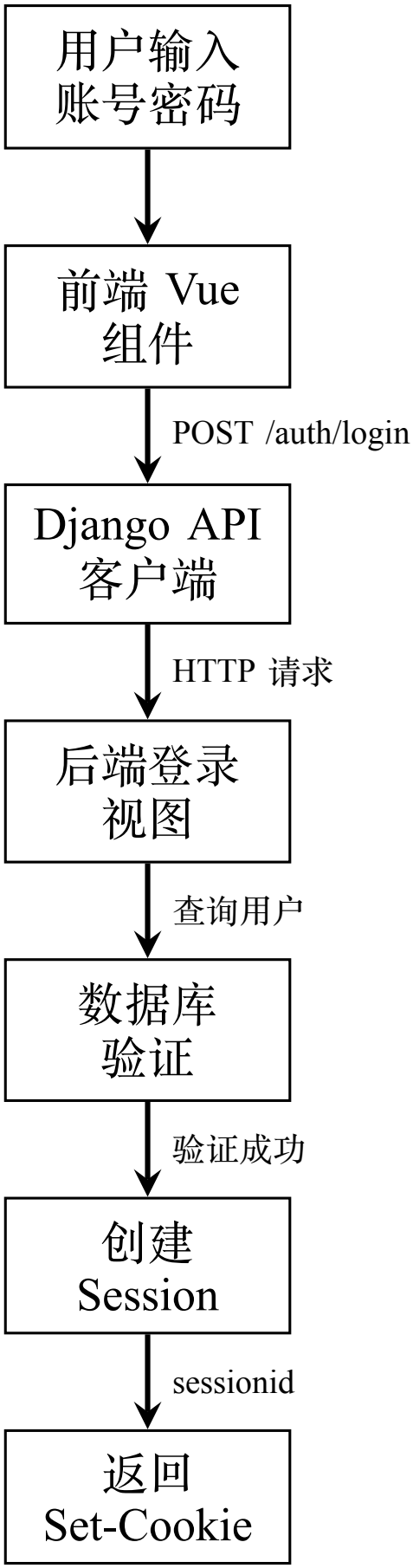

Fig. 1. 用户登录流程图

B. 服务器端登录实现

服务器端登录视图实现：

```python
@csrf_exempt
@allow_methods(["POST"])
def login_view(request: HttpRequest):
    payload = parse_json(request)
    account = payload.get("account")
    password = payload.get("password")
    try:
        user = User.objects.get(account=account,
                                password=password)
    except User.DoesNotExist:
        return api_response(401, "账号或密码错误",
                            status=401)
    # 将用户信息存入 session
    request.session['user_id'] = str(user.user_id)
    request.session['account'] = user.account
    request.session['role'] = user.role
    request.session.set_expiry(86400 * 7)  # 7天过期
    # 确保 session 被保存
    request.session.save()
    return api_response(200, "登录成功",
                        _user_payload(user))
```

Listing 4. 服务器端登录视图

C. *Session* 创建与 *Cookie* 发放

Django Session 中间件处理流程：(1) 从请求 Cookie 读取 `sessionid`；(2) 加载 Session 数据到 `request.session`；(3) 视图函数读写 Session；(4) 响应时，如果 Session 被修改，中间件生成 sessionid、存储到 `django_session` 表、在响应头设置 `Set-Cookie: sessionid=<session_key>`。`request.session.save()` 确保 Session 持久化并触发 Cookie 生成。

IV. Session 存储与用户关联

A. *Session* 数据存储机制

Django 将 Session 数据存储在 `django_session` 表：`session_key`（主键，即 sessionid）、`session_data`（加密后的 Session 数据）、`expire_date`（过期时间）。Session 数据采用 base64 编码的 pickle 格式，经 HMAC 签名确保完整性。

B. 用户身份关联过程

当用户后续请求到达服务器时，认证流程如下：

```python
def require_auth(view_func: Callable) -> Callable:
    """
    认证装饰器：检查用户是否已登录（通过 session）
    """
    @wraps(view_func)
    def wrapper(request: HttpRequest, *args, **kwargs):
        if 'user_id' not in request.session:
            return api_response(401, "未登录，请先登录",
                                status=401)
        return view_func(request, *args, **kwargs)
    return wrapper
```

Listing 5. 认证装饰器

认证流程：(1) 浏览器在请求头自动包含 Cookie: `sessionid=<session_key>`；(2) Session 中间件读取 sessionid，查询 `django_session` 表；(3) 加载 Session 数据到 `request.session`；(4) 视图函数通过 `request.session['user_id']` 获取用户 ID；(5) 装饰器检查 `user_id`，不存在则返回 401。

C. *Session* 生命周期管理

Session 有效期通过 `request.session.set_expiry(86400 * 7)` 设置为 7 天。登出时调用 `request.session.flush()` 删除 Session 记录：

```
def logout_view(request: HttpRequest):
    """登出：清除 session"""
    request.session.flush()
    return api_response(200, "登出成功")
```

Listing 6. 登出实现

V. 密码验证机制

A. 密码存储策略

系统采用明文密码存储。用户模型定义：

```
class User(models.Model):
    class Role(models.TextChoices):
        USER = "user", "user"
        ADMIN = "admin", "admin"

    user_id = models.UUIDField(primary_key=True,
                               default=uuid.uuid4)
    account = models.CharField(max_length=255, unique=True)
    password = models.CharField(max_length=255)
    # 按需求存明文
    role = models.CharField(max_length=10,
                            choices=Role.choices,
                            default=Role.USER)
    created_at = models.DateTimeField(auto_now_add=True)
```

Listing 7. 用户模型

B. 密码验证过程

密码验证逻辑：

```
try:
    user = User.objects.get(account=account,
                            password=password)
except User.DoesNotExist:
    return api_response(401, "账号或密码错误",
                        status=401)
```

Listing 8. 密码验证逻辑

验证步骤：(1) 接收 `account` 和 `password`；(2) 使用 `User.objects.get(account=account, password=password)` 查询；(3) 存在匹配记录则验证成功；(4) 抛出 `User.DoesNotExist` 异常则返回 401。返回统一错误信息” 账号或密码错误” 可防止账号枚举攻击。

VI. 完整认证流程

图 2展示了从登录到后续 API 请求的完整认证流程。

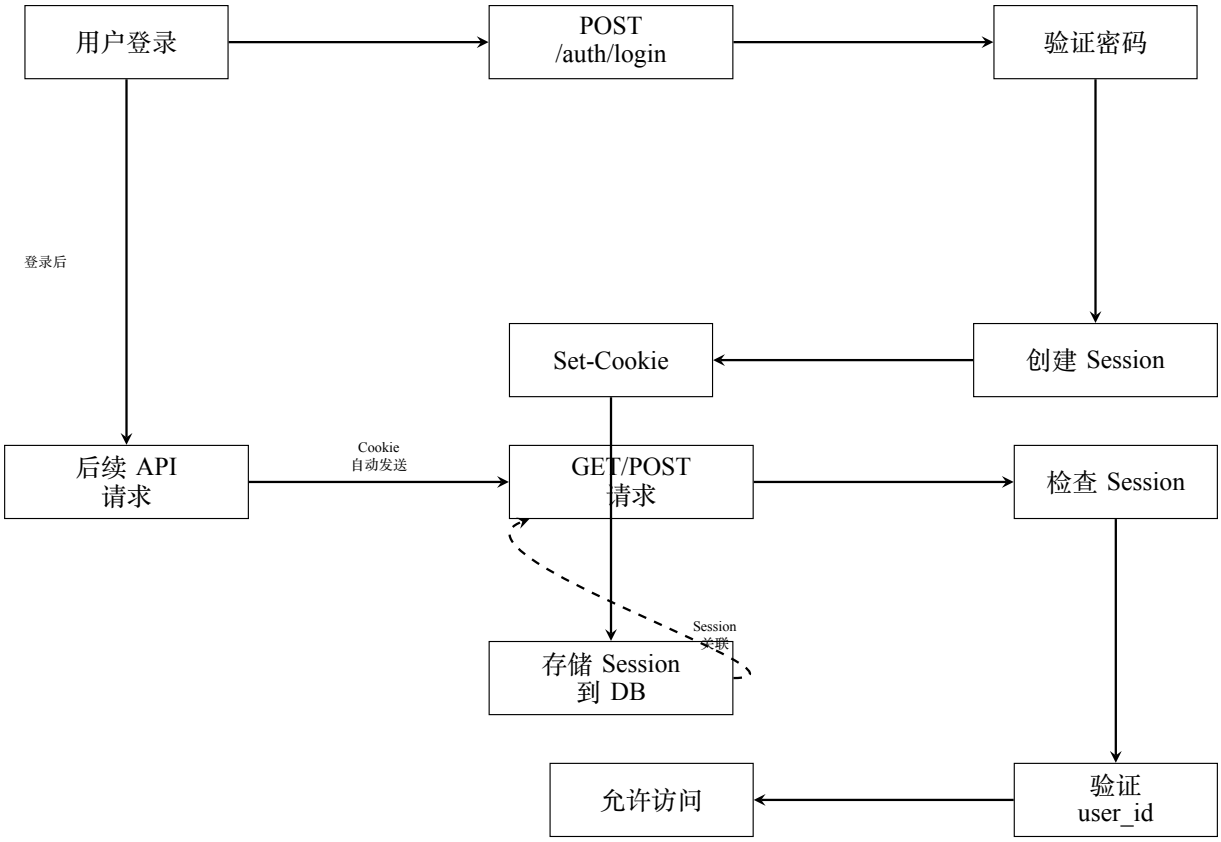

Fig. 2. 完整认证流程示意图

VII. 结论

本文阐述了基于 Django Session 机制的用户认证系统实现，回答了四个核心问题：

1) **Cookie 自动发送**：配置 `withCredentials: true` 后，浏览器在所有 AJAX 请求中自动包含 Session Cookie。
2) **Session 创建流程**：登录成功后，服务器将用户信息存入 Session，Django 中间件自动生成 sessionid 并通过 Set-Cookie 发送给浏览器。
3) **用户身份关联**：Session 数据存储在 `django_session` 表，通过 sessionid 关联。每次请求时，Session 中间件自动加载 Session 数据到 `request.session`。
4) **密码验证**：使用 `User.objects.get(account=account, password=password)` 直接比较验证。

References

[1] Django Documentation. “Sessions.” [Online]. Available: https://docs.djangoproject.com/en/stable/topics/http/sessions/
[2] IETF RFC 6265. “HTTP State Management Mechanism.” [Online]. Available: https://tools.ietf.org/html/rfc6265
[3] Mozilla Developer Network. “Cross-Origin Resource Sharing (CORS).” [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
[4] Axios Documentation. “Request Config.” [Online]. Available: https://axios-http.com/docs/req_config