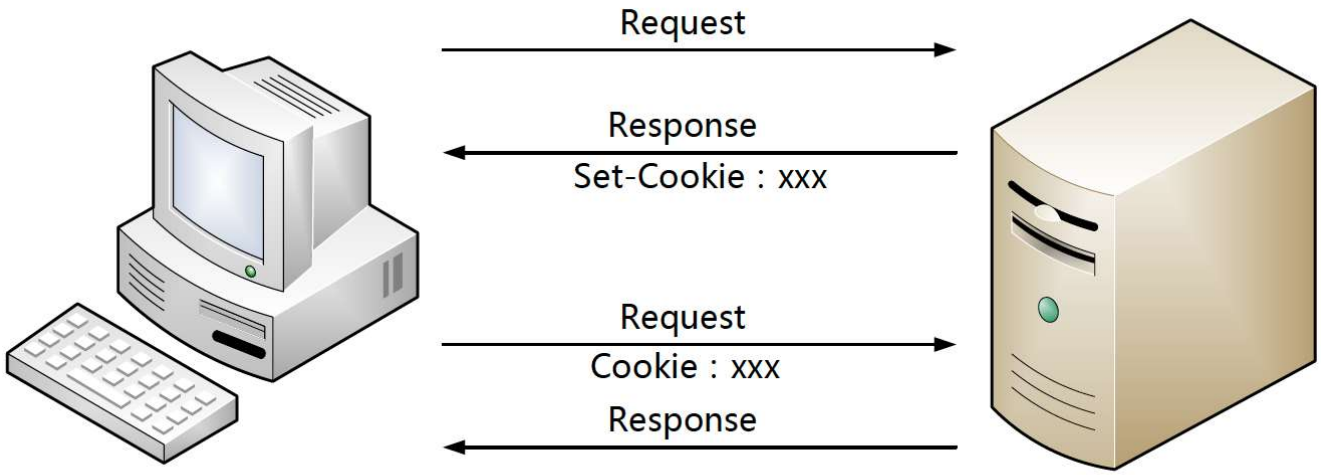
Flask-JWT-Extended中文文档教程

(/user/Qxjjo9SKN

精华 首发 原创 分享 (/post/list-1-1) ·hynever (/user/Qxjjo9SKNod3RzWw6R8b5W) ·2021-11-29 14:12:52 ·3568次点击

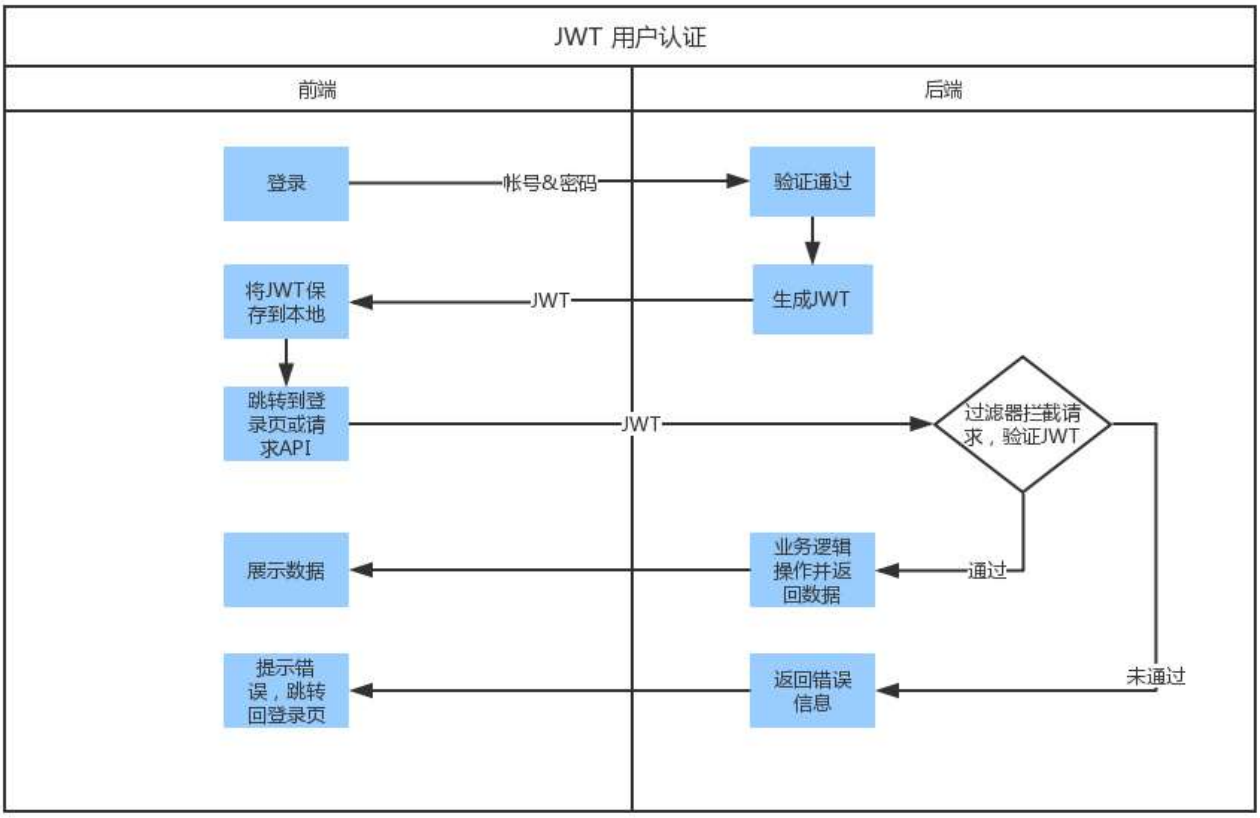
现在前后端分离开发已经是一个web开发者必备的技能了，而通信鉴权问题是前后端分离开发中首要解决的问题，用得最广泛的方式就是 `JWT(JSON Web Token)` 。今天我们来聊一聊为什么需要 `JWT` 、 `JWT` 的原理，以及在 `Flask` 中如何优化又便捷的实现 `JWT` 鉴权。

一、为什么需要JWT:



先抛开 `JWT` ，回顾一下我们传统的网页，是通过 `Cookie` 的方式实现鉴权的，在用户登陆完成后，返回能识别该用户的信息到浏览器的 `Cookie` 中，下一次浏览器请求相同域名的时候，会自动把上次从服务器获取的 `Cookie` 提交上去，从而实现用户鉴权。使用 `Cookie` 的前提是需要同源，也就是渲染页面的域名必须和当前请求的域名相同，否则 `Cookie` 是不会生效的，举个例子，从百度服务器获取的 `Cookie` ，浏览器是不会提交到腾讯服务器的。传统的网页，使用 `Cookie` 是没有问题的，但是如果开发前后端分离项目，就会有一些问题了。前后端分离项目在开发和生产环境中，极大可能是部署在不同的服务器上，也就是域名都不一样，这时候用 `Cookie` 就不太合适。为了解决同源策略的问题，我们就需要使用 `JWT` ，下面再来说一下 `JWT` 的原理，你就能明白 `JWT` 相比 `Cookie` 在前后端分离项目中的优势。

二、JWT原理：



JWT 的原理其实非常简单。用户在登录网站后，我们将能识别该用户的信息，比如就用 user_id ，加上过期时间，以及加密盐，通过某种加密算法生成一个加密后的字符串，这个字符串就是 JWT ，然后返回给前端。下次前端再发送请求的时候，把这个 JWT 携带上来，然后服务器再把这个 JWT 进行解密，得到 user_id 和过期时间，如果在过期时间内，并且 user_id 是正确的，那么就能识别此请求是哪个用户了。通过 JWT ，前后端分离项目部署在不同服务器上，也可以正常通信。这里有个小细节，就是可以在生成 JWT 后，把 JWT 在服务器上也存储一份，这样可以实现单点登陆功能。

三、Flask中使用 JWT ：

在 Python 中，有一个独立的 JWT 包，叫做 PyJWT ： https://pyjwt.readthedocs.io/，我们可以直接使用他来生成 JWT 和验证。但是在 Flask 中，我们可以通过 Flask-JWT-Extended 来实现 JWT 功能，因为他封装了使用方式，以及一些属性和装饰器，用起来更爽。

1. 安装Flask-JWT-Extended：

使用 pip 开始使用此扩展的最简单方法：

```
$ pip install flask-jwt-extended
```

如果要使用非对称（公钥/私钥）密钥签名算法，可以使用包含 asymmetric_crypto 包的 flask-jwt-extended 。

```
$ pip install flask-jwt-extended[asymmetric_crypto]
```

请注意，如果你使用的是 ZSH·（也可能是其他 shell`），则需要对括号进行转义。

```
$ pip install flask-jwt-extended\[asymmetric_crypto\]
```

2. 基本使用：

使用 `flas-jwt-extended` 非常简单，下面来说一下使用步骤。

2.1. 初始化jwt对象：

首先从 `flask_jwt_extended` 中导入 `JWTManager` ，然后创建一个对象，代码如下：

```
# exts.py
from flask_jwt_extended import JWTManager
...
jwt = JWTManager()
```

接着在 `app.py` 中导入 `jwt` 对象，然后进行初始化，代码如下：

```
# app.py
from exts import jwt
...
app = Flask(__name__)
# 这一步一定要设置，会被flask-jwt-extended当做加密的盐
app.config['SECRET_KEY'] = "secret key"
...
jwt.init_app(app)
```

2.2. 生成 `jwt` ：

我们可以使用 `create_access_token` 来创建一个 `token` ，在创建的时候，需要传入能识别此用户的 `identity` 。示例代码如下：

```
from flask_jwt_extended import create_access_token
@app.route("/login", methods=["POST"])
def login():
    username = request.json.get("username", None)
    password = request.json.get("password", None)
    if username != "test" or password != "test":
        return jsonify({"msg": "用户名或密码错误"}), 401

    access_token = create_access_token(identity=username)
    return jsonify(access_token=access_token)
```

2.3. 验证 `jwt` ：

如果某个视图函数必须要验证完 `jwt` 后才能访问，那么可以使用 `jwt_required` 装饰器。然后在视图函数中，使用 `get_jwt_identity` 获取之前创建 `jwt` 时候传入的 `identity` 参数。示例代码如下：

```
@app.route("/protected", methods=["GET"])
@jwt_required()
def protected():
    # Access the identity of the current user with get_jwt_identity
    username = get_jwt_identity()
    return jsonify(logged_in_as=username), 200
```

3. 前端携带JWT：

在获取了 `jwt` 后，下次重新访问的时候，就可以携带JWT来访问一些需要授权的请求了。访问的方式是在请求头中添加 `Authorization` 参数，示例代码如下：

```
http GET :5000/protected Authorization:"Bearer {{jwt}}"
```

上述命令，是使用httpie工具的http命令，往 `http://127.0.0.1:5000/protected` 发送了一个请求，并且在请求头中携带了 `Authorization` 参数，参数的值是 `Bearer jwt` ，其中 `Bearer` 也是一个固定的写法，我们只需要把服务器返回的 `jwt` 设置到 `Bearer` 后面即可。

当然，也可以把 `jwt` 设置到 `cookie` 中， `Body` 中，甚至是请求URL的参数中，但设置在请求头中实际上是最方便的，只要在请求方法中封装好，调用起来非常方便。

4. 过期时间：

默认 `flask-jwt-extended` 的过期时间是15分钟，如果想要自己设置过期时间，那么可以在 `app.config` 中设置 `JWT_ACCESS_TOKEN_EXPIRES` ，比如设置30分钟，那么可以如下代码实现：

```
app.config['JWT_ACCESS_TOKEN_EXPIRES'] = datetime.timedelta(minutes=30)
```

5. 刷新JWT：

`JWT` 的有效时间，不能设置得太长，因为如果被黑客截获了，那么将造成不可估量的影响。我们一般设置15-30分钟后过期，这样就会存在一个问题，用户每隔15-30分钟就要登录一次，显然是不合理的，我们需要在 `JWT` 过期之前刷新 `JWT` 。 关于如何使用刷新 `JWT` ，有多种选择。首先可以在前端存储访问令牌的过期时间，每次发出 `API` 请求时，首先检查当前访问 `JWT` 是否接近或已经过期，并根据需要刷新。这种方法非常简单，在大多数情况下都可以正常工作，但请注意，如果前端时钟明显偏离，可能会遇到问题。

另一种方法是使用你的访问令牌发出 `API` 请求，然后检查结果以查看它是否有效。如果请求的结果是一条错误消息，b表示 `JWT` 已过期，这时候我们使用一个 `刷新JWT` 生成新的 `普通JWT` 并使用新的 `普通JWT` 重做请求。无论前端的时钟如何，这种方法都可以工作，但用起来也确实麻烦一点。

当您的前端不是网站（移动、仅 api 等）时，推荐使用刷新令牌。示例代码如下：

```
from datetime import timedelta

from flask import Flask
from flask import jsonify

from flask_jwt_extended import create_access_token
from flask_jwt_extended import create_refresh_token
from flask_jwt_extended import get_jwt_identity
from flask_jwt_extended import jwt_required
from flask_jwt_extended import JWTManager

app = Flask(__name__)

app.config["JWT_SECRET_KEY"] = "super-secret"
# 设置普通JWT过期时间
app.config["JWT_ACCESS_TOKEN_EXPIRES"] = timedelta(hours=1)
# 设置刷新JWT过期时间
app.config["JWT_REFRESH_TOKEN_EXPIRES"] = timedelta(days=30)
jwt = JWTManager(app)


@app.route("/login", methods=["POST"])
def login():
    access_token = create_access_token(identity="example_user")
    refresh_token = create_refresh_token(identity="example_user")
    return jsonify(access_token=access_token, refresh_token=refresh_token)


# 使用刷新JWT来获取普通JWT
@app.route("/refresh", methods=["POST"])
@jwt_required(refresh=True)
def refresh():
    identity = get_jwt_identity()
    access_token = create_access_token(identity=identity)
    return jsonify(access_token=access_token)


@app.route("/protected", methods=["GET"])
@jwt_required()
def protected():
    return jsonify(foo="bar")


if __name__ == "__main__":
    app.run()
```

以后在遇到 `普通JWT` 过期了，那么可以再使用 `刷新JWT` 来重新获取 `普通JWT` 。

关于 `flask-jwt-extended` 的讲解就在这里，学会这些，您在前后端分离项目中的鉴权，将没有任何问题。读者如果还要了解更多，可以再仔细阅读 `flask-jwt-extended` 的官方文档：
`https://flask-jwt-extended.readthedocs.io/en/stable/` 。

希望本文对您有帮助，欢迎点赞收藏，我将更有动力发布更多干货文章！