使用 Flask 设计 RESTful 的认证

今天我将要展示一个简单，不过很安全的方式用来保护使用 Flask 编写的 API，它是使用密码或者令牌认证的。

示例代码

本文使用的代码能够在 github 上找到: REST-auth 。

用户数据库

为了让给出的示例看起来像真实的项目，这里我将使用 Flask-SQLAlchemy 来构建用户数据库模型并且存储到数据库中。

用户的数据库模型是十分简单的。对于每一个用户，username 和 password_hash 将会被存储:

```
class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key = True)
    username = db.Column(db.String(32), index = True)
    password_hash = db.Column(db.String(128))
```

出于安全原因，用户的原始密码将不被存储，密码在注册时被散列后存储到数据库中。使用散列密码的话，如果用户数据库不小心落入恶意攻击者的手里，他们也很难从散列中解析到真实的密码。

密码 **决不能** 很明确地存储在用户数据库中。

密码散列

为了创建密码散列，我将会使用 PassLib 库，一个专门用于密码散列的 Python 包。

PassLib 提供了多种散列算法供选择。custom_app_context 是一个易于使用的基于 sha256_crypt 的散列算法。

User 用户模型需要增加两个新方法来增加密码散列和密码验证功能:

```
from passlib.apps import custom_app_context as pwd_context

    class User(db.Model):
        # ...

        def hash_password(self, password):
            self.password_hash = pwd_context.encrypt(password)

        def verify_password(self, password):
            return pwd_context.verify(password, self.password_hash)
```

hash_password() 函数接受一个明文的密码作为参数并且存储明文密码的散列。当一个新用户注册到服务器或者当用户修改密码的时候，这个函数将被调用。

verify_password() 函数接受一个明文的密码作为参数并且当密码正确的话返回 True 或者密码错误的话返回 False。这个函数当用户提供和需要验证凭证的时候调用。

你可能会问如果原始密码散列后如何验证原始密码的？

散列算法是单向函数，这就是意味着它们能够用于根据密码生成散列，但是无法根据生成的散列逆向猜测出原密码。然而这些算法是具有确定性的，给定相同的输入它们总会得到相同的输出。PassLib 所有需要做的就是验证密码，通过使用注册时候同一个函数散列密码并且同存储在数据库中的散列值进行比较。

用户注册

在本文例子中，一个客户端可以使用 POST 请求到 /api/users 上注册一个新用户。请求的主体必须是一个包含 username 和 password 的 JSON 格式的对象。

Flask 中的路由的实现如下所示:

```
@app.route('/api/users', methods = ['POST'])
def new_user():
    username = request.json.get('username')
    password = request.json.get('password')
    if username is None or password is None:
        abort(400) # missing arguments
    if User.query.filter_by(username = username).first() is not None:
        abort(400) # existing user
    user = User(username = username)
    user.hash_password(password)
    db.session.add(user)
    db.session.commit()
    return jsonify({ 'username': user.username }), 201, {'Location': url_for('get_user', id =
user.id, _external = True)}
```

这个函数是十分简单地。参数 username 和 password 是从请求中携带的 JSON 数据中获取，接着验证它们。

如果参数通过验证的话，新的 User 实例被创建。username 赋予给 User，接着使用 hash_password 方法散列密码。用户最终被写入数据库中。

响应的主体是一个表示用户的 JSON 对象，201 状态码以及一个指向新创建的用户的 URI 的 HTTP 头信息：Location。

注意：get_user 函数可以在 github 上找到完整的代码。

这里是一个用户注册的请求，发送自 curl:

```
$ curl -i -X POST -H "Content-Type: application/json" -d '{"username":"miguel","password":"python"}'
http://127.0.0.1:5000/api/users
HTTP/1.0 201 CREATED
Content-Type: application/json
Content-Length: 27
Location: http://127.0.0.1:5000/api/users/1
Server: Werkzeug/0.9.4 Python/2.7.3
Date: Thu, 28 Nov 2013 19:56:39 GMT

{
  "username": "miguel"
}
```

需要注意地是在真实的应用中这里可能会使用安全的的 HTTP (譬如：HTTPS)。如果用户登录的凭证是通过明文在网络传输的话，任何对 API 的保护措施是毫无意义的。

基于密码的认证

现在我们假设存在一个资源通过一个 API 暴露给那些必须注册的用户。这个资源是通过 URL: /api/resource 能够访问到。

为了保护这个资源，我们将使用 HTTP 基本身份认证，但是不是自己编写完整的代码来实现它，而是让 Flask-HTTPAuth 扩展来为我们做。

使用 Flask-HTTPAuth，通过添加 login_required 装饰器可以要求相应的路由必须进行认证:

```
from flask.ext.httpauth import HTTPBasicAuth
auth = HTTPBasicAuth()

@app.route('/api/resource')
@auth.login_required
def get_resource():
    return jsonify({ 'data': 'Hello, %s!' % g.user.username })
```

但是，Flask-HTTPAuth 需要给予更多的信息来验证用户的认证，当然 Flask-HTTPAuth 有着许多的选项，它取决于应用程序实现的安全级别。

能够提供最大自由度的选择(可能这也是唯一兼容 PassLib 散列)就是选用 verify_password 回调函数，这个回调函数将会根据提供的 username 和 password 的组合的，返回 True(通过验证) 或者 Flase(未通过验证)。Flask-HTTPAuth 将会在需要验证 username 和 password 对的时候调用这个回调函数。

verify_password 回调函数的实现如下:

```
@auth.verify_password
def verify_password(username, password):
    user = User.query.filter_by(username = username).first()
    if not user or not user.verify_password(password):
        return False
    g.user = user
    return True
```

这个函数将会根据 username 找到用户，并且使用 verify_password() 方法验证密码。如果认证通过的话，用户对象将会被存储在 Flask 的 g 对象中，这样视图就能使用它。

这里是用 curl 请求只允许注册用户获取的保护资源:

```
$ curl -u miguel:python -i -X GET http://127.0.0.1:5000/api/resource
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 30
Server: Werkzeug/0.9.4 Python/2.7.3
Date: Thu, 28 Nov 2013 20:02:25 GMT

{
  "data": "Hello, miguel!"
}
```

如果登录失败的话，会得到下面的内容:

```
$ curl -u miguel:ruby -i -X GET http://127.0.0.1:5000/api/resource
HTTP/1.0 401 UNAUTHORIZED
Content-Type: text/html; charset=utf-8
Content-Length: 19
WWW-Authenticate: Basic realm="Authentication Required"
Server: Werkzeug/0.9.4 Python/2.7.3
Date: Thu, 28 Nov 2013 20:03:18 GMT

Unauthorized Access
```

这里我再次重申在实际的应用中，请使用安全的 HTTP。

基于令牌的认证

每次请求必须发送 username 和 password 是十分不方便，即使是通过安全的 HTTP 传输的话还是存在风险，因为客户端必须要存储不加密的认证凭证，这样才能在每次请求中发送。

一种基于之前解决方案的优化就是使用令牌来验证请求。

我们的想法是客户端应用程序使用认证凭证交换了认证令牌，接下来的请求只发送认证令牌。

/

令牌是具有有效时间，过了有效时间后，令牌变成无效，需要重新获取新的令牌。令牌的潜在风险在于生成令牌的算法比较弱，但是有效期较短可以减少风险。

有很多的方法可以加强令牌。一个简单的强化方式就是根据存储在数据库中的用户以及密码生成一个随机的特定长度的字符串，可能过期日期也在里面。令牌就变成了明文密码的重排，这样就能很容易地进行字符串对比，还能对过期日期进行检查。

更加完善的实现就是不需要服务器端进行任何存储操作，使用加密的签名作为令牌。这种方式有很多的优点，能够根据用户信息生成相关的签名，并且很难被篡改。

Flask 使用类似的方式处理 cookies 的。这个实现依赖于一个叫做 itsdangerous 的库，我们这里也会采用它。

令牌的生成以及验证将会被添加到 User 模型中，其具体实现如下:

```
from itsdangerous import TimedJSONWebSignatureSerializer as Serializer

class User(db.Model):
    # ...

    def generate_auth_token(self, expiration = 600):
        s = Serializer(app.config['SECRET_KEY'], expires_in = expiration)
        return s.dumps({ 'id': self.id })

    @staticmethod
    def verify_auth_token(token):
        s = Serializer(app.config['SECRET_KEY'])
        try:
            data = s.loads(token)
        except SignatureExpired:
            return None # valid token, but expired
        except BadSignature:
            return None # invalid token
        user = User.query.get(data['id'])
        return user
```

/

generate_auth_token() 方法生成一个以用户 id 值为值，'id' 为关键字的字典的加密令牌。令牌中同时加入了一个过期时间，默认为十分钟(600 秒)。

验证令牌是在 verify_auth_token() 静态方法中实现的。静态方法被使用在这里，是因为一旦令牌被解码了用户才可得知。如果令牌被解码了，相应的用户将会被查询出来并且返回。

API 需要一个获取令牌的新函数，这样客户端才能申请到令牌:

```
@app.route('/api/token')
@auth.login_required
def get_auth_token():
    token = g.user.generate_auth_token()
    return jsonify({ 'token': token.decode('ascii') })
```

注意：这个函数是使用了 auth.login_required 装饰器，也就是说需要提供 username 和 password。

剩下来的就是决策客户端怎样在请求中包含这个令牌。

HTTP 基本认证方式不特别要求 usernames 和 passwords 用于认证，在 HTTP 头中这两个字段可以用于任何类型的认证信息。基于令牌的认证，令牌可以作为 username 字段，password 字段可以忽略。

这就意味着服务器需要同时处理 username 和 password 作为认证，以及令牌作为 username 的认证方式。verify_password 回调函数需要同时支持这两种方式:

```
@auth.verify_password
def verify_password(username_or_token, password):
    # first try to authenticate by token
    user = User.verify_auth_token(username_or_token)
    if not user:
        # try to authenticate with username/password
        user = User.query.filter_by(username = username_or_token).first()
        if not user or not user.verify_password(password):
            return False
    g.user = user
    return True
```

新版的 verify_password 回调函数会尝试认证两次。首先它会把 username 参数作为令牌进行认证。如果没有验证通过的话，就会像基于密码认证的一样，验证 username 和 password。

如下的 curl 请求能够获取一个认证的令牌:

```
$ curl -u miguel:python -i -X GET http://127.0.0.1:5000/api/token
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 139
Server: Werkzeug/0.9.4 Python/2.7.3
Date: Thu, 28 Nov 2013 20:04:15 GMT

{
  "token":
"eyJhbGciOiJIUzI1NiIsImV4cCI6MTM4NTY2OTY1NSwiaWF0IjoxMzg1NjY5MDU1fQ.eyJpZCI6MX0.XbOEFJkhjHJ5uRINh2JA1BPz

}
```

现在可以使用令牌获取资源:

```
$ curl -u
eyJhbGciOiJIUzI1NiIsImV4cCI6MTM4NTY2OTY1NSwiaWF0IjoxMzg1NjY5MDU1fQ.eyJpZCI6MX0.XbOEFJkhjHJ5uRINh2JA1BPzX
 -i -X GET http://127.0.0.1:5000/api/resource
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 30
Server: Werkzeug/0.9.4 Python/2.7.3
Date: Thu, 28 Nov 2013 20:05:08 GMT

{
  "data": "Hello, miguel!"
}
```

需要注意的是这里并没有使用密码。

OAuth 认证

当我们讨论 RESTful 认证的时候，OAuth 协议经常被提及到。

那么什么是 OAuth？

OAuth 可以有很多的含义。最通常就是一个应用程序允许其它应用程序的用户的接入或者使用服务，但是用户必须使用应用程序提供的登录凭证。我建议阅读者可以浏览 OAuth 了解更多知识。