Implementar un JWT (JSON Web Token) Authorization Server en web2py

Rodolfo López (rlopezt@uvaq.edu.mx)

Objetivo

Implementar un Authorization Server en web2py para generar tokens válidos con el estándar JSON Web Token (JWT) para controlar el acceso a APIS REST expuestas en NondeJS, adicionalmente esto nos permitirá liberar al servido de la gestión sesiones realizando peticiones stateless.

Desarrollo

El control de accesos utilizado actualmente por el Framework web2py es Control de Acceso Basado en Roles (RBAC), el cual es una técnica para restringir el acceso al sistema a usuarios autorizados a través de roles, permitiendo asignar los privilegios mínimos necesarios a dichos usuarios, por lo que se pueden aprovechar estas funcionalidades de control de accesos para generar tokens JWT de manera segura basado en los roles.

El control de accesos nativo en web2py nos permitira configurar “claims” personalizados y basado en roles para restringir el acceso y ejecución de procesos y otras acciones en APIs.

Aunque en este ejemplo se definen “claims” consireando solo el usuario y su rol asignado, se pueden definir más parámetros y ámbitos para definir los tipos de recursos protegidos a los que el cliente puede acceder.

Para conocer más de la estructura de JWT puede visitar https://jwt.io/introduction/

Para conocer más sobre los claims utilizados por JWT puede visitar https://www.iana.org/assignments/jwt/jwt.xhtml

A continuación se presenta la arquitectura propuesta.

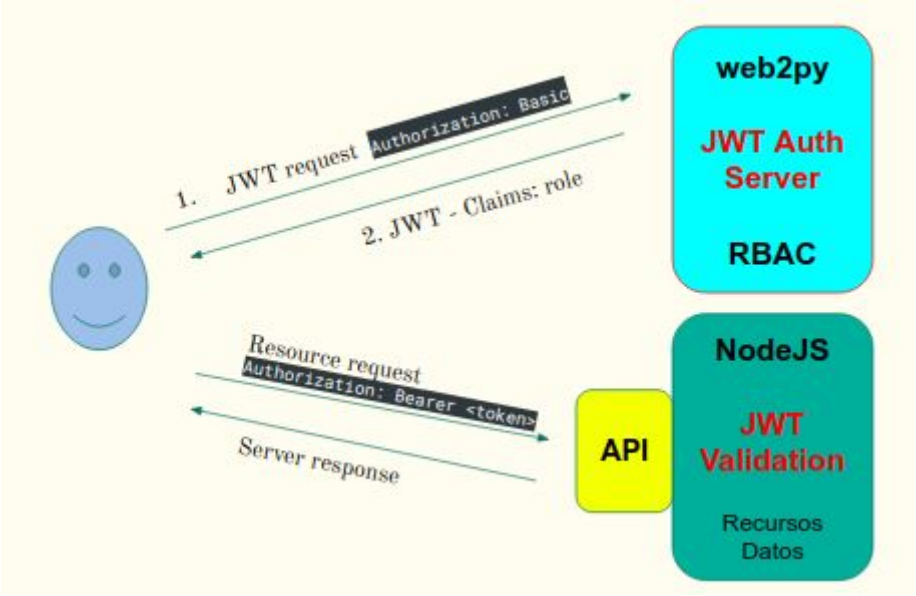

Figura 1. Arquitectura Authentication Server JWT

web2py soporta distintos tipos de autenticación como LDAP, OpenID, PAM, OAuth2.0, entre otros, pero en este caso utilizaremos una autenticación básica HTTP el cual es un mecanismo sencillo de autenticación, donde las credenciales se envían dentro del Header HTTP Authorization codificadas en base64 de la forma username:password.

```
GET /index.html HTTP/1.0
Host: basic.example.com
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

El servidor responde con un 200 OK si el usuario es autenticado de manera exitosa o de otra manera con un código 40X.

Implementación del Authorization Server

Utilizaremos la estructura de Control de Acceso Basada en Roles (RBAC) implementada en el framework web2py python, dicho control está basado en la siguiente estructura.

- `auth_user` almacena el nombre del usuario, dirección de correo electrónico, contraseña y estado (pendiente de registro, aceptado, bloqueado)
- `auth_group` almacena los grupos o roles para usuarios en una estructura muchos-a-muchos. Por defecto, cada usuario pertenece a su propio grupo, pero un usuario puede estar incluido en múltiples grupos, y cada grupo contener múltiples usuarios. Un grupo es identificado por su rol y descripción.

- `auth_membership` enlaza usuarios con grupos en una estructura muchos-a-muchos.
- `auth_permission` enlaza grupos con permisos. Un permiso se identifica por un nombre y opcionalmente, una tabla y un registro. Por ejemplo, los miembros de cierto grupo pueden tener permisos "update" (de actualización) para un registro específico de una tabla determinada.
- `auth_event` registra los cambios en las otras tablas y el acceso otorgado a través de CRUD a objetos controlados con RBAC.
- `auth_cas` se usa para el Servicio Central de Autenticación (CAS). Cada aplicación web2py es un proveedor de CAS y puede opcionalmente consumir el servicio CAS.

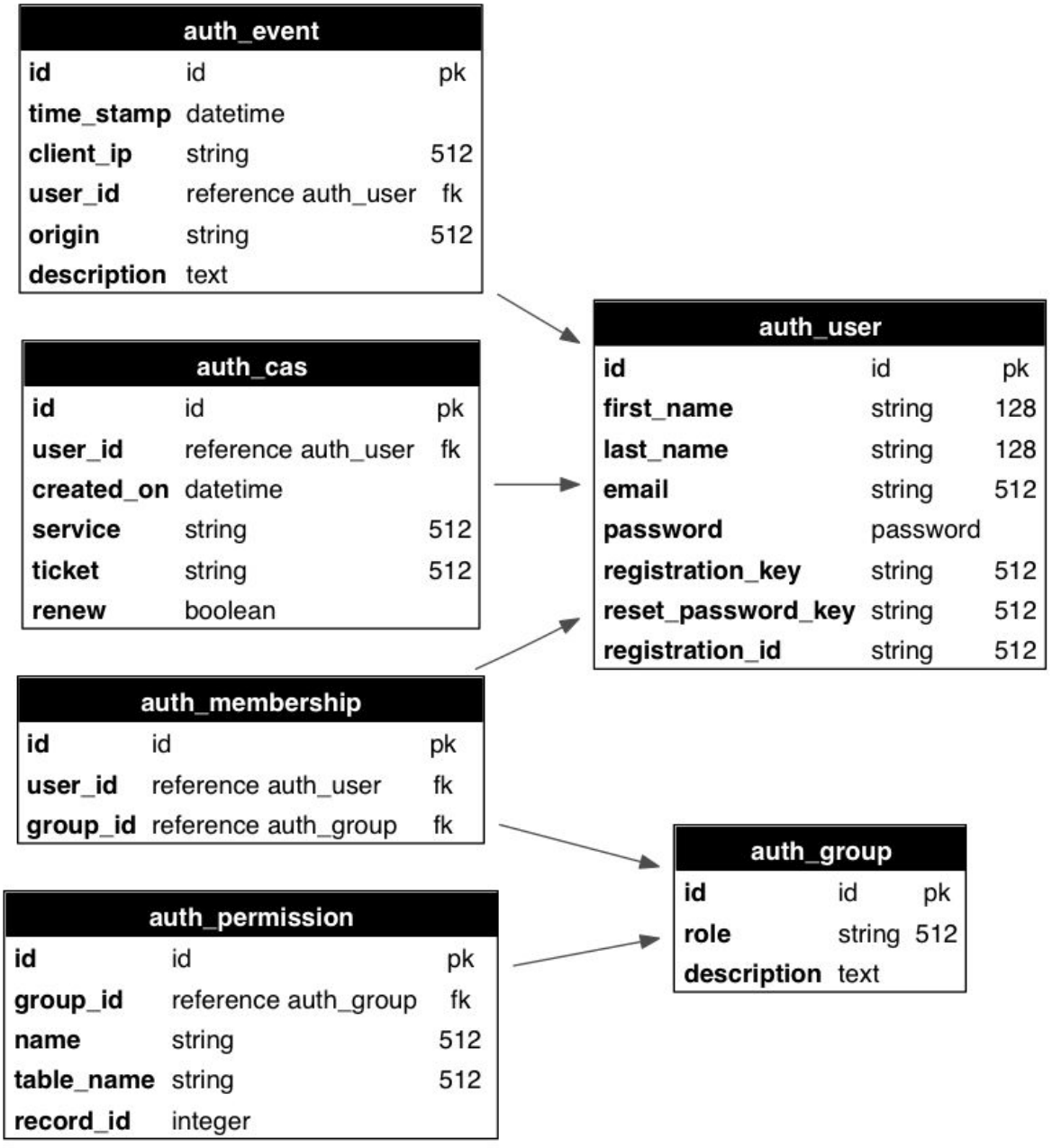

Figura 2. Módelo de Datos de Control de Accesos web2py

Para habilitar autenticación básica se debe importar la siguiente clase en el model de web2py.

from gluon.contrib.login_methods.basic_auth import basic_auth

Y habilitar las siguientes configuraciones en el model auth policy

auth.settings.allow_basic_login = True #activate basic auth
auth.settings.login_methods = [basic_auth()] #force to use only basic auth

Una vez realizadas las configuraciones anteriores en el model, se define la siguiente función en el controller, dicha función será utilizada para enviar los tokens a los usuarios autenticados exitosamente.

```
@auth.requires_login()
def api_get_jwt():
    session.forget()
    if not request.env.request_method == 'GET': raise HTTP(403)
    import jwt
    import json
    import datetime
    secret = 'secret'
    role = db((db.auth_membership.id==auth.user.id) &
(db.auth_membership.group_id==db.auth_group.id)).select(db.auth_group.role).first().role
    sub = auth.user.username
    exp = datetime.datetime.utcnow() + datetime.timedelta(seconds=60)
    payload = {'sub':sub, 'role': role, 'exp': exp}
    encoded_jwt = jwt.encode(payload, secret, algorithm='HS256' )
    return encoded_jwt
```

Algunos puntos a resaltar son:

Se utilizan las librerias python pyjwt (pip install pyjwt)
La variable “secret” se utiliza para firmar el token.
El rol del usuario se obtiene consultando la tabla auth_membership.
Se define un tiempo de expiración del password en la variable exp.
Las peticiones se reciben por HTTP GET pero se puede indicar otro método HTTP.

Lo anterior expone una API que es accedida solo por usuarios previamente registrados y autenticados por medio de HTTP Basic Auth, lo cual se muestra con la siguiente petición.

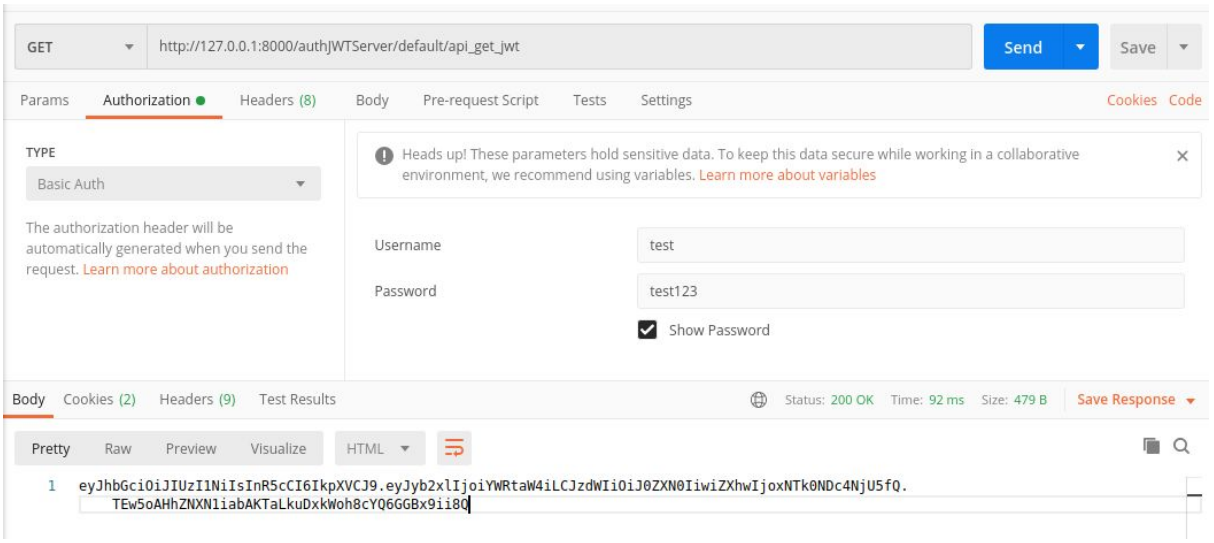

Figura 3. Petición HTTP GET al Authentication Server para obtener token JWT

Una vez obtenido el token JWT podemos realizar la petición de recursos a través de APIs expuestas en un entorno de ejecución NodeJS express, el cual validará la correcta estructura y firma del token recibido, así como el rol asignado al usuario, para permitir el acceso solo a las APIs autorizadas.

Para realizar este ejemplo definimos la siguiente función, la cual será accedida solo por usuarios con un token válido.

```
const express = require('express');
const jwt = require('jsonwebtoken');
const app = express();
const router = express.Router();

router.post('/protected', function(req, res)
      {
            var token = req.headers['authorization']
            if(!token){
                  res.status(401).send({error: "No token found"})
                  return}

            token = token.replace('Bearer ', '')
            var decoded = jwt.decode(token);

            jwt.verify(token, 'secret', function(err, user)
                  {
```

```
                    if (err)
                    {
                        res.status(401).send({error: 'Invalid Token'})
                    }
                    else if (decoded['role'] == 'admin')
                    {
                            res.send({message: 'Authorized'});
                    }
                    else
                    {
                        res.send({message: 'NOT Authorized'})
                    }
                }
            )
        }
);
```

Algunos puntos a resaltar son:

Se utiliza la libreria 'jsonwebtoken' y ‘express’.
En la función “verify” se requiere de la llave utilizada para firmar el token jwt.verify(token, 'secret', function(err, user).
Con la variable decoded obtenemos los claims decodificados para validar el rol, en este caso solo permite acceso a usuarios con el rol “admin”.

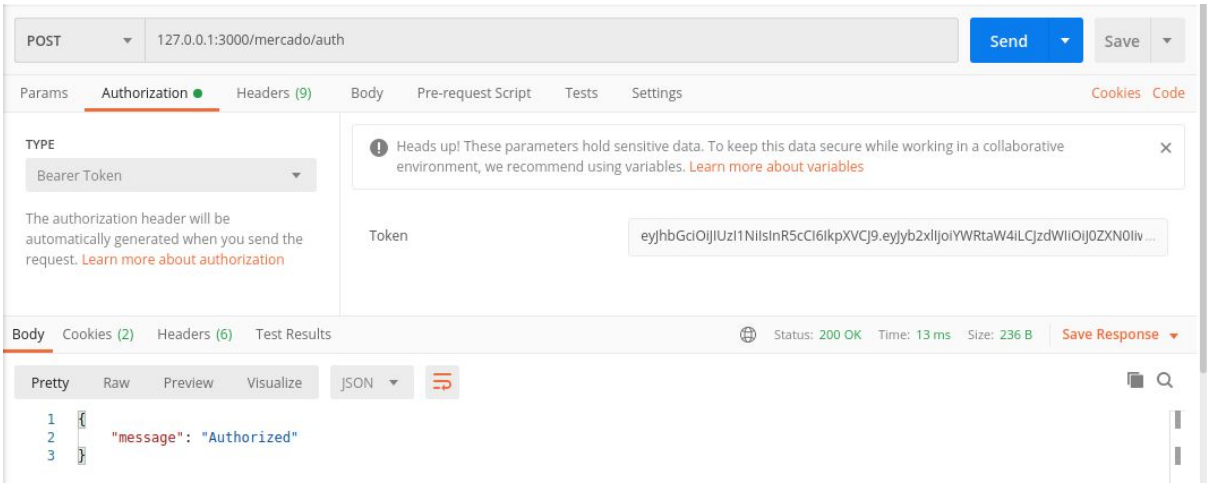

Figura 4. Petición HTTP POST para acceder a API

Conclusiones

Implementando el Authorization Server para obtener token JWT en web2py es una buena opción ya que se aprovecha la estructura definida nativamente de control de accesos basada en roles RBAC lo cual nos permite tener una capa de seguridad para los usuarios autenticados exitosamente, y utilizar los atributos y roles de los usuarios para definir

“claims” que tengan la información necesaria para consumir APIs en el entorno de ejecución NodeJS express de manera segura.