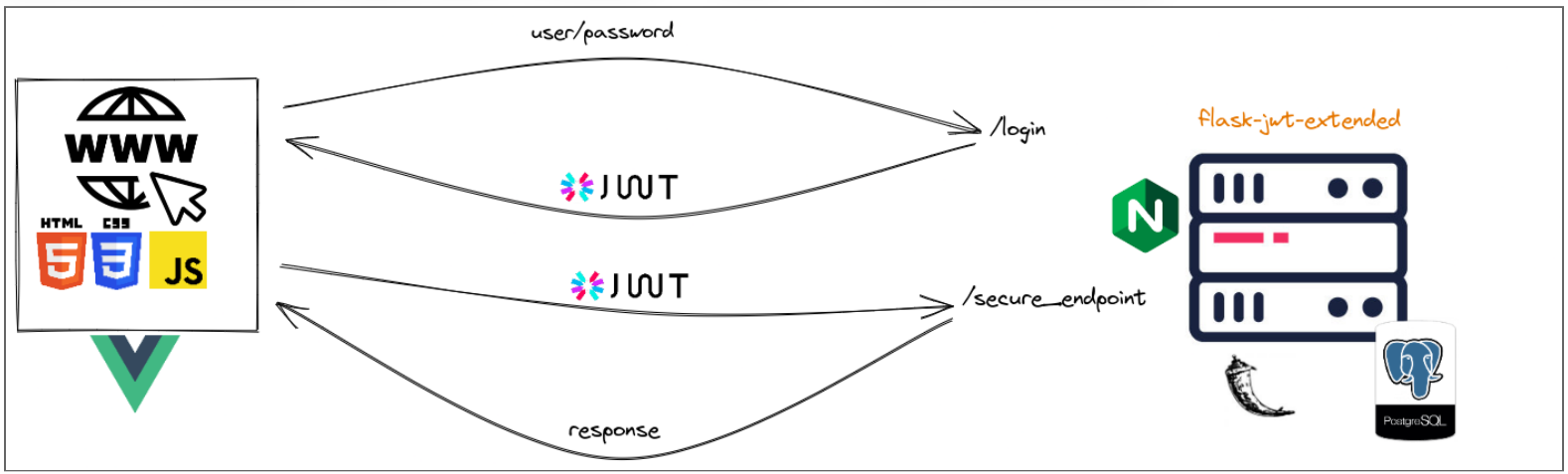
PROYECTO DE SOFTWARE

AUTENTICACIÓN CON JWT: FLASK <-JWT-> VUE (COOKIES)



EN FLASK:

flask-jwt-extended

Extensión de Flask que básicamente nos permite trabajar con JWT mediante:

- create_access_token() para crear el JSON Web Tokens.
- jwt_required() para proteger rutas.
- get_jwt_identity() para obtener la identidad contenida en el JWT.

FLASK-JWT-EXTENDED

Instalación:

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

o

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

Si vamos un algoritmo de criptografía asimétrica (clave pública/privada).

¿CÓMO MANDAMOS EL JWT? ¿Y DÓNDE LO GUARDAMOS?

Se debe configurar con **JWT_TOKEN_LOCATION**:

```
app.config["JWT_TOKEN_LOCATION"] = ["headers", "cookies",
"json", "query_string"]
```

HEADERS

- El token es recibido al loguearse y debe ser almacenado en alguna parte (por ejemplo **localStorage**).
- En cada requerimiento debe ser enviado por mediante un **header**.
- Eliminando el token del **localStorage** terminaría la sesión.

```
async function login() {
  const response = await fetch('/login_without_cookies',
{method: 'post'});
  const result = await response.json();
  localStorage.setItem('jwt', result.access_token);
}
function logout() {
  localStorage.removeItem('jwt');
}
async function makeRequestWithJWT() {
  const options = {
    method: 'post',
    headers: {
      Authorization: `Bearer
${localStorage.getItem('jwt')}`,
    }
  };
  const response = await fetch('/protected', options);
  const result = await response.json();
```

```
  return result;
}
```

JSON BODY

Muy similar a Headers, excepto que se envía el JWT en el body del requerimiento.

```
async function makeRequestWithJWT() {
  const options = {
    method: 'post',
    body: JSON.stringify({access_token:
localStorage.getItem('jwt')}),
    headers: {
      'Content-Type': 'application/json',
    },
  };
  const response = await fetch('/protected', options);
  const result = await response.json();
  return result;
}
```

Lo malo es que **NO** se puede utilizar en requerimiento sin body como **GET** o **HEAD**.

QUERY STRING

En este caso se envía el JWT en el string de la query. **Totalmente desaconsejado**, el token queda expuesto totalmente, aún en una conexión SSL.

```
async function makeRequestWithJWT() {
  const jwt = localStorage.getItem('jwt')
  const response = await fetch(`/protected?jwt=${jwt}`,
{method: 'post'});
  const result = await response.json();
  return result;
}
```

COOKIES

Una buena manera de manejar JWTs en navegadores es con **cookies**, nos da algunos beneficios sobre **headers**:

- Se pueden configurar para ser enviados únicamente por HTTPS: **JWT_COOKIE_SECURE**. Esto evita que el JWT sea enviado en una conexión insegura.
- Son enviadas como cookies **HTTP ONLY**, lo cual previene que sea manipulada por JS y por lo tanto mitiga los ataques de XSS.
- Flask puede manejar el refresco de los tokens cercanos a expirar.
- Se cuenta con una 2da cookie para prevenir Cross Site Request Forgery (CSRF): **X-CSRF-TOKEN** que debe enviarse ante cada requerimiento.

```
async function makeRequestWithJWT() {
  const options = {
    method: 'post',
    credentials: 'same-origin',
    headers: {
      'X-CSRF-TOKEN': getCookie('csrf_access_token'),
    },
```

```
  };
  const response = await fetch('/protected', options);
  const result = await response.json();
  return result;
}
```

¿CÓMO VERIFICAMOS EL JWT?

- Primero debemos definir con qué algoritmo firmar el JWT.

```
app.config["JWT_ALGORITHM"] = ["HS256"]
```

- Básicamente tenemos 2 grandes opciones:
  - **Cifrado simétrico** (la opción por defecto): se usa un secreto en el servidor utilizado para verificar los tokens. Se configura en **JWT_SECRET_KEY**.
  - **Cifrado asimétrico**: se tienen un par de claves publica/privada en el servidor, esto nos da la ventaja de poder desde el cliente verificar el token que nos envía el servidor con su clave pública. Se configuran **JWT_PRIVATE_KEY** y **JWT_PUBLIC_KEY**.

LOGIN Y CREACIÓN DE TOKEN

Es un chequeo de login nomal con el agregado de la creación del token indicando la identidad a utilizar **create_access_token** y seteando las cookies de la respuesta **set_access_cookies**.

```
@auth_blueprint.post('/login_jwt')
def login_jwt():
  data = request.get_json()
  email = data['email']
  password = data['password']
  user = auth.find_user_by_email_and_pass(email, password)

  if user:
    access_token = create_access_token(identity=user.id)
    response = jsonify()
    set_access_cookies(response, access_token)
    return response, 201
  else:
    return jsonify(message="Unauthorized"), 401
```

Nota: *se pueden agregar más datos al token, pero recuerden que al ser una cookie seteada con* ***HTTP ONLY*** *no vamos a poder manipularla con JS.*

ENDPOINT SEGUROS

El JWT es verificado automáticamente en los endpoints seguros (@jwt_required).

```
@auth_blueprint.get('/user_jwt')
@jwt_required( )
def user_jwt():
current_user = get_jwt_identity()
user = auth.get_user_by_id(current_user)
response = jsonify(user)
return response, 200
```

- La función **get_jwt_identity()** nos devuelve la identidad del usuario para el cuál fue creado el JWT.

LOGOUT

El logout simplemente elimina la cookie del cliente con **unset_jwt_cookies**.

```
@auth_blueprint.get('/logout_jwt')
@jwt_required( )
def logout_jwt():
  response = jsonify()
  unset_jwt_cookies(response)
  return response, 200
```

PARA SEGUIR LEYENDO:

- Refreshing Tokens: **https://flask-jwt-extended.readthedocs.io/en/stable/refreshing_tokens/**
- CSRF Tokens: **https://flask-jwt-extended.readthedocs.io/en/stable/options/#cross-site-request-forgery-options**

Y EN VUE.JS?

- Como estamos utilizando **cookies**, el token se va a obtener con el **login**. Aunque, recuerden que no podemos leer los datos dentro del token (**HTTP ONLY**).
- Pero... si el login es satisfactorio (un 200), ya **tenemos acceso a los endpoints protegidos** para obtener la información necesaria del usuario en forma segura.

LOGIN EN VUE.JS

- En la app de ejemplo vamos a utilizar **vuex** para tener un estado comaprtido que maneje la información de login.

```
const state = {
  user: {},
  isLoggedIn: false
};
```

- Junto con las acciones que realizan el login, la obtención de la información segura y el logout.

```
const actions = {
    async loginUser({ dispatch }, user) {
        await apiService.post('/auth/login_jwt', user)
        await dispatch('fetchUser')
    },
    async fetchUser({ commit }) {
        await apiService.get('/auth/user_jwt')
            .then(({ data }) => commit('setUser', data))
    },
    async logoutUser({ commit }) {
        await apiService.get('/auth/logout_jwt');
        commit('logoutUserState');
```

```
    }
};
```

PETICIONES CON CREDENCIALES

- En este caso vamos a utilizar **axios** para realizar las peticiones (aunque se puede utilizar Fetch directamente).

```
import axios from 'axios';

const apiService = axios.create({
    baseURL: 'http://localhost:5000/',
    withCredentials: true,
    xsrfCookieName: 'csrf_access_token'
});

export { apiService };
```

*Lo importante es configurar que los requerimientos **lleven los tokens de credenciales**, sino no van a ser enviados.*

DEMO

REFERENCIAS:

- Flask-JWT-Extended: **https://flask-jwt-extended.readthedocs.io/**
- PyJWT algoritmos soportados: **https://pyjwt.readthedocs.io/en/latest/algorithms.html**
- Cookie Based Authentication with Flask and Vue.js:
  - Parte 1 - Flask: **https://fareedidris.medium.com/cookie-based-authentication-using-flask-and-vue-js-part-1-c625a530c157**
  - Parte 2 - Vue.js: **https://fareedidris.medium.com/cookie-based-authentication-using-flask-and-vue-js-part-2-bd2b4754546**

FIN