Projet d’API et refonte du système de sessions pour Muonium

**Introduction**

Actuellement Muonium ne possède pas d’API à proprement parler, seulement différentes routes utilisées par le client web lors de requêtes Ajax (en GET ou POST), sans utiliser le format JSON pour les données. De plus, le système actuel n’est pas documenté, et pas standardisé. Ce qui rend difficile voire impossible la mise en œuvre d’applications ou de clients pour Muonium par d’autre développeurs étant donné le contexte propriétaire.

Cela rend aussi difficile le développement de l’application mobile et de bureau, en plus de compliquer l’évolution.

Le back-end de Muonium est écrit en PHP et organisé selon la structure MVC, les changements à prévoir de ce côté ne sont pas énormes, si ce n’est pour les vues (du moins celles qui ne retournent pas des pages entières) qui devront retourner des données au format JSON en lieu et place de code HTML ou de texte brut. Concernant les contrôleurs, il faudra peut-être créer de nouvelles routes, prendre en compte d’autres méthodes http que POST ou GET et adapter les données reçues au nouveau format.

Nous utilisons les sessions PHP et la base de données utilisée est MariaDB, il n’est pas prévu pour le moment d’y toucher.

**Implications**

Nous souhaitons donc créer une API RESTful pour Muonium, et dans un second temps réorganiser le client selon une architecture MVC et passer les vues en HTML (pages complètes) du côté client. A terme, les échanges entre le client et le serveur seront uniquement au format JSON.

Les sessions en PHP ne sont pas possible pour une API RESTful car il y a la contrainte de ne pas stocker l'état du client sur le serveur et chaque requête doit contenir toutes les informations nécessaires pour la comprendre.

JWT (JSON Web Token) permet de répondre à cette problématique, nous l'avons choisi pour sa simplicité. Le principe : une fois authentifié, le serveur génère un token contenant les informations de la session et le signe selon une clé privée. Le client envoie une requête avec le token et le serveur va le décoder et vérifier l'intégrité du contenu.

**Cahier des charges**

Les différentes contraintes à respecter dans notre cas :

- Créer une API RESTful sans sessions côté serveur, chaque requête doit contenir toutes les informations nécessaires.
- Utiliser les différentes méthodes (HTTP  GET, POST, PUT, DELETE, HEAD, PATCH) et adapter le routing/les contrôleurs en conséquence.
- Chaque instance doit être indépendante, pouvoir se connecter sur différents appareils, différents onglets. Se déconnecter/changer de compte sur une instance sans implications sur les autres. Gérer les différentes sessions.
- Pouvoir invalider une session (déconnexion/expiration/changement de mot de passe)
- Multi-threading pour le back-end
- Système de sessions très rapide
- Faciliter le load balancing

**Avantages et inconvénients des sessions PHP**

+ Simple à mettre en place, solution mature et efficace, très utilisé
+ Peu de données qui transitent
+ Le client ne peut pas voir leur contenu
+ Délai d'expiration renouvelé à chaque requête
- Le stockage dans un cookie ne permet pas d'ouvrir plusieurs sessions dans une même fenêtre
- Pas adapté pour une API REST
- Ne fonctionne pas si les cookies sont bloqués par le client
- Sensible au vol du cookie de session et à la faille CSRF
- Pas adapté pour voir le nombre de sessions actives
- Problème de scalabilité, load balancing plus compliqué à mettre en place

*"There are some solutions to fix/avoid this: Load balance with session affinity Use a centralized session like memcached, redis, databases and other ways PHP provides"*
https://www.codementor.io/byjg/using-json-web-token-jwt-as-a-php-session-axeuqbg1m

**Avantages et inconvénients de JWT**

+ Le client ne peut pas voir leur contenu (signé)
+ Adapté pour les API
+ Fonctionne avec les cookies bloqués s'il n'est pas stocké dans un cookie
+ Peut être stocké dans différents endroits (localStorage, sessionStorage, cookie)
+ Meilleure scalabilité
+ N'utilise pas le système de fichiers du serveur
+ Prévu pour l'authentification unique (SSO)
+ Pas vulnérable CSRF si pas stocké dans un cookie
- Ne fonctionne pas entre différents domaines
- Plus de données qui transitent
- Délai d'expiration non renouvelé
- Impossible d'invalider un token avant expiration
- La clé secrète peut être compromise

Dans les inconvénients de JWT, le renouvellement du délai d'expiration et l'impossibilité d'invalider le token avant expiration posent problème par rapport à nos besoins.

Concernant le multithreading, PHP n'est pas le meilleur langage pour cela mais au vu de l'infrastructure prévue, du code s'exécutera bien sur plusieurs threads, et c'est le cas lors de l'upload et le download car le client envoi des requêtes au serveur via Ajax qui est asynchrone. On peut néanmoins le prévoir dans le code PHP pour certains cas spécifiques, au cas par cas, par exemple si dans une même méthode on effectue un traitement et qu'on est bloqué par autre chose.

Pour nos inconvénients, il suffirait de renouveler le token à chaque requête pour mettre à jour l'expiration et tenir une whitelist/blacklist en base de données des tokens à valider/invalider.

Nous utiliserons un système de base de données exclusivement pour l'authentification, avec de très hautes performances et adapté à cet usage : Redis.

https://redis.io/

Afin de mettre en œuvre JWT et Redis avec PHP, nous allons utiliser deux librairies :

- PHP Jwt (JSON Web Tokens) (https://github.com/firebase/php-jwt)
- Predis (https://github.com/nrk/predis)

**Tests préalables réalisés**

Mettre une valeur dans le sessionStorage et actualiser : on récupère bien cette valeur.

Page dans un nouvel onglet : le sessionStorage est vide, on met une nouvelle valeur mise dedans.

Retour sur l'onglet précédent (qui n'a pas été fermé) : on a bien la valeur initiale en actualisant.

De même pour l'autre onglet.

Fermer et rouvrir l'onglet : la valeur est perdue.

Nous avons donc la confirmation que c'est bien indépendant.

*Alors pourquoi ce n'est pas le cas sur Muonium qui store la cek dans le sessionStorage ?*

Car si on ouvre un nouvel onglet on perd la cek mais on garde la session PHP, or la cek étant vide on va être redirigé vers Login, jusqu'ici c'est normal mais la session PHP va être détruite lors de la déconnexion et donc commune au navigateur avec les différents onglets (cookie). En gardant les infos de session côté client avec vérification token côté serveur on n'a pas ce souci.

**Nouveau système de sessions**

Voici à quoi pourrait ressembler notre fichier *composer.json* :

Composer.json

```
{
	"require": {
		"firebase/php-jwt": "^4.0",
		"phpmailer/phpmailer": "~6.0",
		"predis/predis": "1.1"
	},

	"autoload" : {
		"psr-0": {
			"application\\": "src",
			"config\\": "src",
			"library\\": "src"
		}
	}
}
```

A propos de l'invalidation des tokens :

« *You have two options to invalidate all tokens of a particular user:*

1. *Keep a list (in the database, using a Cache provider, etc) of all tokens. This is probably the easiest way to do it (I had to do this for one project), but it kind-of defeats the whole "sessionless" purpose of JWT.*
2. *Store the timestamp of when the user's password was last changed. When the user uses a token, compare the iat field of the token with the timestamp of the password change. If the iat is before the password change, blacklist the token and log the user out. »*

https://github.com/tymondesigns/jwt-auth/issues/1263

Pour nos besoins, nous avons défini un délai d'expiration (exp) de 20mn et nous utiliserons HS384 pour signer le token.

Exemple de génération :

```
private function buildToken() {
        $secretKey = \config\secretKey::get();
        $tokenId    = base64_encode(mcrypt_create_iv(32));
        $issuedAt   = time();
        $notBefore  = $issuedAt + 10;
        $exp     = $notBefore + 60;
        $serverName = 'localhost';

        $data = [
            'iat'  => $issuedAt, // Issued at: time when the token
was generated
            'jti'  => $tokenId,  // Json Token Id: an unique
identifier for the token
            'iss'  => $serverName,  // Issuer
            'nbf'  => $notBefore,   // Not before
            'exp'  => $exp,         // Expire
            'data' => [          // Data related to the signer user
                'id'   => 1,
                'username' => 'toto'
            ]
        ];

        $jwt = \Firebase\JWT\JWT::encode(
            $data,      //Data to be encoded in the JWT
            $secretKey, // The signing key
            'HS384'     // Algorithm used to sign the token, see
https://tools.ietf.org/html/draft-ietf-jose-json-web-algorithms-
40#section-3
        );

        $jwt = json_encode(['jwt' => $jwt]);
           // Put in a view...
           /* */
    }
```

Pour le décoder :

```
try{
        $decoded = JWT::decode($jwt,  $secretKey, 'HS384');
} catch(\Exception $e){
         echo 'Caught exception: ',  $e->getMessage(), "\n";
}
```

Pour différencier les différentes méthodes http, on utilisait httpMethodsData.php dans l'exemple d'intégration JWT avec MUI, voir pour la reprendre et intégrer au niveau du routing/du contrôleur.
Dans une requête, envoyer token dans le header Authorization avec Bearer

**Problème : renouveler token avant expiration**

- ⇨ Comme expliqué plus haut, nous pouvons à chaque requête faire expirer le token et en renvoyer un nouveau si la date d'expiration est toujours valide.

Mais on y perdrait en performances et plus d'I/O avec la base de données.

Une amélioration possible :

Générer un token seulement quand on arrive à la moitié (ou plus) du temps d'expiration, cela permet un gain en performances, le seul défaut : si exp = 20mn, on ne fait rien pendant 9mn puis une requête : le token n'est pas renouvelé car on n'est pas à la moitié. On ne fait rien ensuite pendant 12mn, le token est expiré. Ce n'est à priori pas gênant tant que exp/2 n'est pas trop court.
exp serait de 1 mois sur l'application mobile.

Il faut mettre en place un middleware sur le client pour mettre à jour le token à chaque réponse du serveur s'il en renvoie un.

**Problème : Utilisation d'un ancien token toujours valide car date pas dépassée**

- ⇨ Stocker sur Redis les tokens pas expirés (whitelist) avec l'id user, date création, et éventuellement des infos sur l'appareil.
- ⇨ Permet de voir, de terminer toutes les sessions d'un coup et de les faire expirer.

On perd l'aspect de ne dépendre d'aucune base de données de JWT mais on perd ses défauts comme expliqué plus haut.

Déconnexion/renouvellement token/changement mdp : Suppression du token dans Redis.

La plupart du temps, les personnes qui implémentent JWT avec Redis, stockent seulement les tokens expirés (blacklist) et effacent au bout d'un certain temps mais cela ne permet pas de voir les différentes sessions, et de les terminer (excepté celle sur l'appareil)

Exemple :
« token :xxx », « true »
« token :xxx:id », « 123456 »
“ token:xxx:created”, “1518018397”
…

“id:xxx”, “1,2,3,4” /* different tokens valides */

Autres propositions :

- Salt pour le hash du token: possible en le stockant dans la db.
- Clé privée dynamique : il faudrait la stocker en db, donc est-ce que ça vaut réellement le coup ?

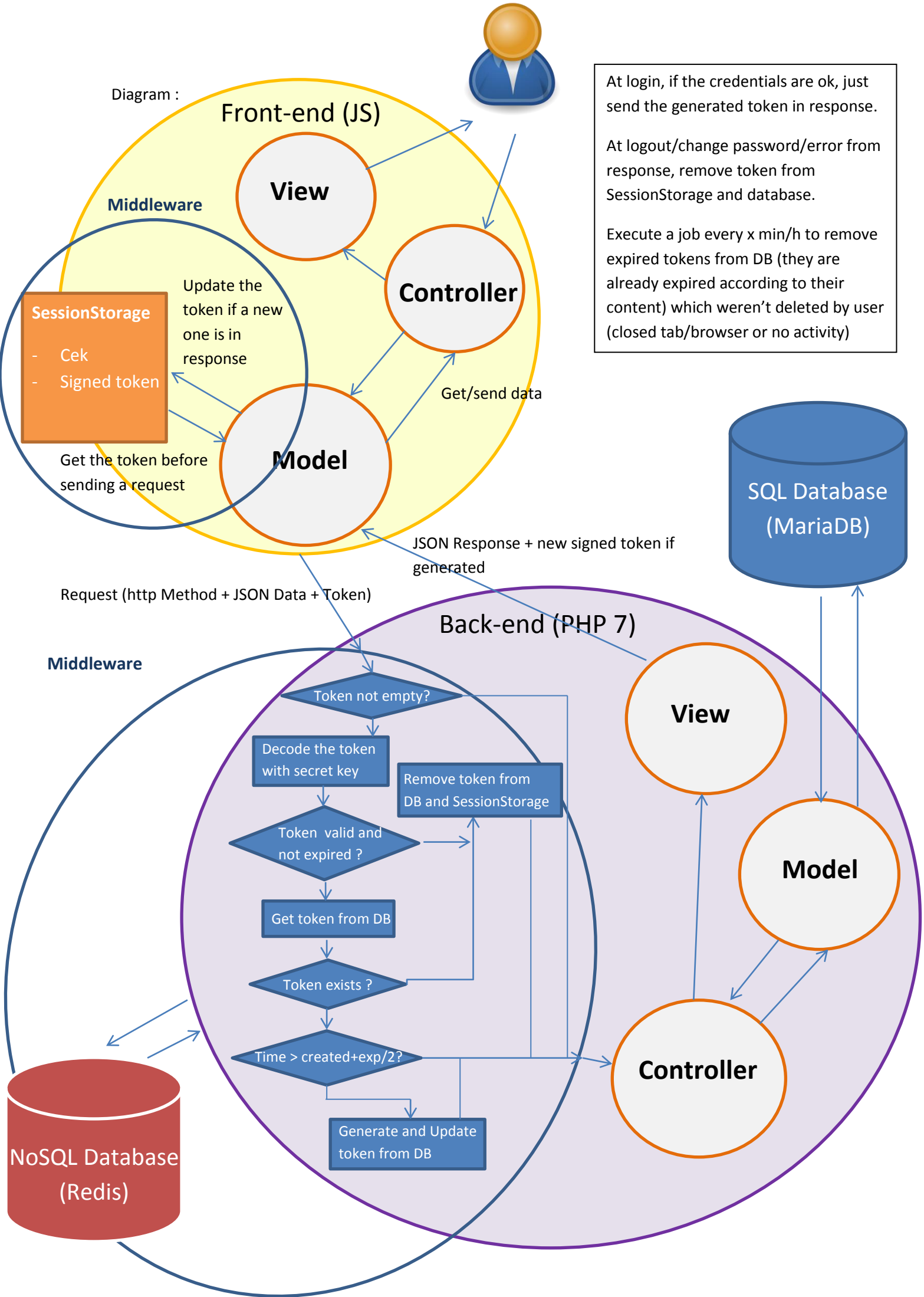
Diagram :
Front-end (JS)
View
Middleware
Controller
SessionStorage
- Cek
- Signed token
Update the token if a new one is in response
Get/send data
Get the token before sending a request
Model
At login, if the credentials are ok, just send the generated token in response.
At logout/change password/error from response, remove token from SessionStorage and database.
Execute a job every x min/h to remove expired tokens from DB (they are already expired according to their content) which weren't deleted by user (closed tab/browser or no activity)
SQL Database (MariaDB)
JSON Response + new signed token if generated
Request (http Method + JSON Data + Token)
Back-end (PHP 7)
Middleware
Token not empty?
Decode the token with secret key
Remove token from DB and SessionStorage
Token valid and not expired ?
Get token from DB
Token exists ?
Time > created+exp/2?
Generate and Update token from DB
View
Model
Controller
NoSQL Database (Redis)