GSB avec Angular2, partie 5

Un peu de sécurité

Code fourni

Le nouveau service REST : https://github.com/patricegrand/restGSBsecu
Le code de l'application initiale : https://github.com/patricegrand/GSBAngular2_V5.0

Introduction

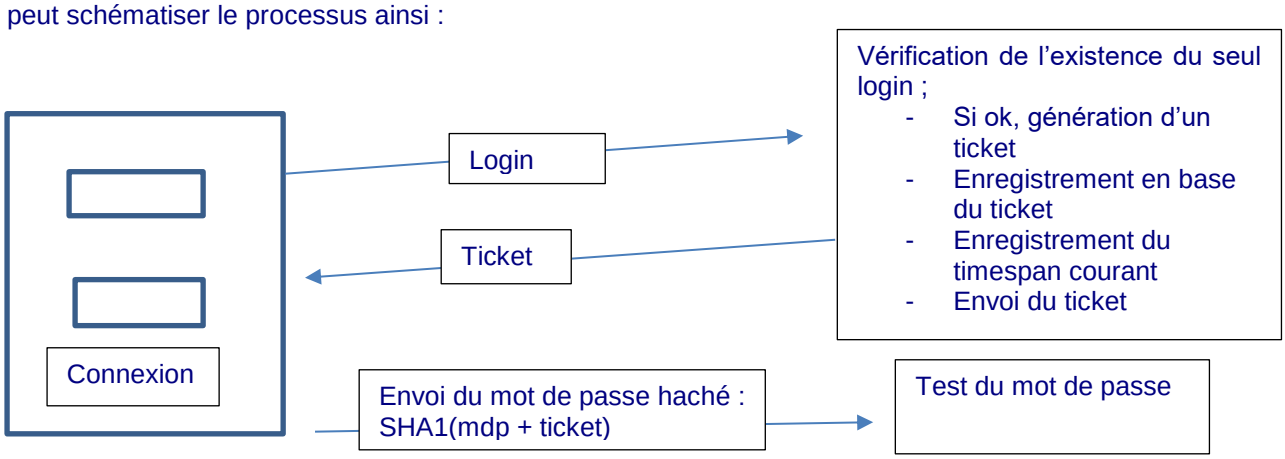
Nous allons aborder, pour cette dernière partie, l'aspect de la sécurité pour une application utilisant un service REST.
L'utilisation de services REST rend une application vulnérable dans la mesure où « quiconque » peut avec un simple navigateur mener des requêtes sans passer par l'application ; il est donc nécessaire de mettre en place un dispositif sécurisé. Sécurisé par le contrôle strict de l'auteur de la requête, par la circulation hachée des données sensibles, par la limitation dans le temps de la validation de la connexion.
L'**annexe** 1 rappelle les notions de cryptage, hachage, « salage » que vous devez connaître au préalable.

Présentation du dispositif de sécurisation

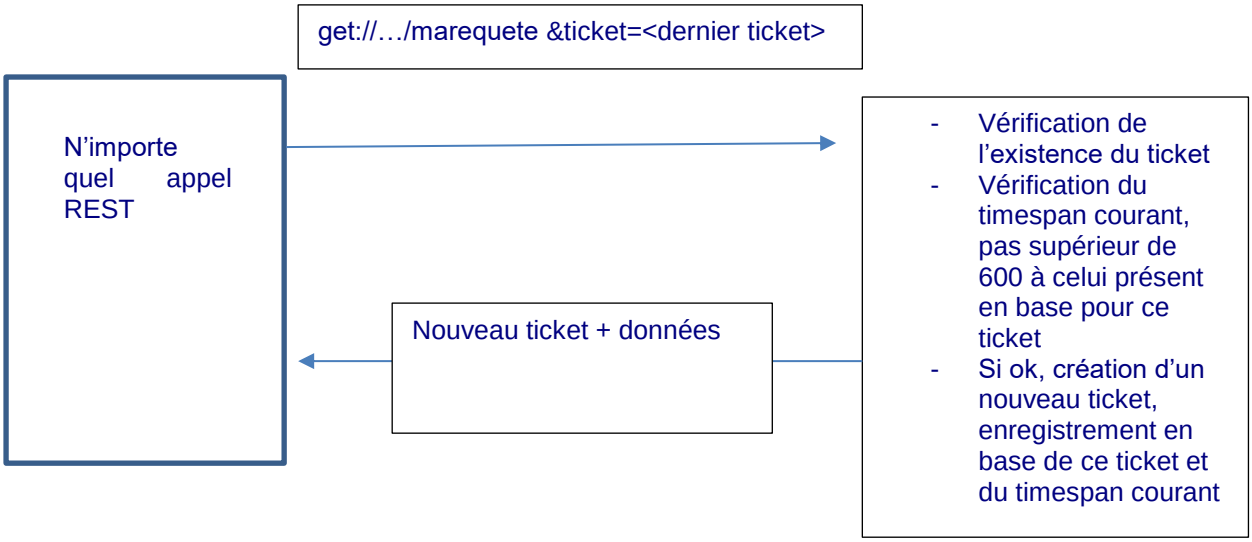
La sécurisation mise en œuvre dans notre cas va respecter certaines règles.

1) Le mot de passe ne circulera jamais en clair
2) Le mot de passe ne circulera qu'à la connexion (haché avec un *sel*)
3) A chaque connexion validée sera associé un ticket personnel dont la validité sera limitée dans le temps ; c'est ce ticket temporaire qui identifiera l'auteur de la requête
4) Chaque requête générera un nouveau ticket personnel
5) Le ticket sera sauvegardé en base, ainsi que le moment où il a été émis

On peut schématiser le processus ainsi :



Remarque : plutôt que d'enregistrer le *DateTime* courant, il est préférable et d'usage usage (plus pratique en fait) d'enregistrer son *timespan* qui est le nombre de secondes écoulées entre le 1 janvier 1970 et le *DateTime* courant ; les comparaisons de *DateTime* sont ainsi plus simples.

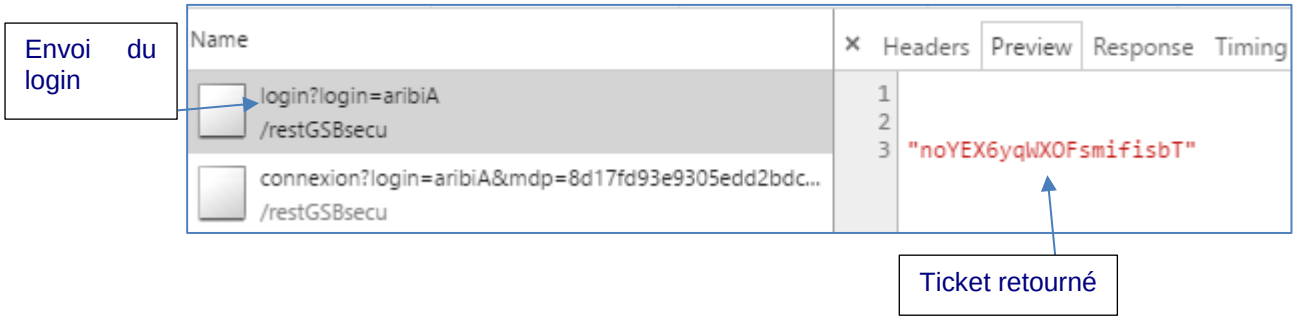


Gestion de l'authentification

1) *Côté serveur, modification du service REST pour la connexion*

Nous ne rentrerons pas dans les détails ; le code vous est fourni et nous allons juste commenter le mécanisme de connexion et la génération du ticket. Lors de la connexion, le schéma présenté plus haut montre que deux appels REST sont effectués :

a) Test du login (unique dans la base) :



Le ticket retourné dans la réponse est obtenu grâce à la méthode *getLogin* associée à l'appel REST :

```
300     private function getLogin($args){
301         $login = $args['login'];
302         $ticket = $this->pdo->existeLogin($login);
303         if($ticket != null){
304               $this->data = json_encode($ticket);
305         }
306         else{
307             $this->data="";
308             $this->codeRetour=400;
309         }
310 
311     }
```

La méthode *existeLogin* de la classe PDO recherche le ticket correspondant au login :

```
54          public function existeLogin($login){
55                  $req = "select id from visiteur where login = :login";
56                  $stm = self::$monPdo->prepare($req);
57                  $stm->bindParam(':login', $login);
58                  $stm->execute();
59                  $laLigne = $stm->fetch();
60                  if($laLigne!=null){
61                     $ticket = $this->setTicket($laLigne['id']);
62                     return $ticket;
63                  }
64                  return NULL;
65 
66          }
```

Dans le cas où le login est trouvé, un nouveau ticket est créé (appel de *setTicket*) :

```
197         public function setTicket($visiteur){
198                 $time = time();
199                 $val="azertyuiopqsdfghjklmwxcvbn1234567890AZERTYUIOPQSDFGHJKLMWXCVBN";
200                 $ticket ="";
201                 for($i=0;$i<20;$i++)
202                         $ticket .= $val[rand(0,strlen($val)-1)];
203                 $req = "update visiteur set timespan = ".$time. ", ticket ='".$ticket."' where ";
204                 $req.= " id ='".$visiteur."' or ticket = '". $visiteur ."'";
205                 $rs = self::$monPdo->query($req);
206                 return $ticket;
```

- lignes 199 à 202, génération d'un ticket aléatoire sur 20 caractères
- ligne 203, enregistrement en base de ce ticket et du timespan courant.
- cette requête peut être lancée avec l'id du visiteur (lors de la connexion) ou avec la valeur du ticket.
- ce ticket sera retourné à l'application qui consomme ce service (l'application Angular)

Remarque : le risque de collision de deux tickets identiques est infiniment faible…

b) Test du mot de passe

C'est le rôle du deuxième appel REST :

Cet appel va demander la vérification du login et du mot de passe haché (mdp+ticket) ; si ce mot de passe est validé, un nouveau ticket est émis et retourné :

```
312     private function connexion($args){
313         $login = $args['login'];
314         $mdp = $args['mdp'];
315         $ticket = $this->pdo->verifierVisiteur($login, $mdp);
316         if($ticket != NULL){
317                     $newTicket = $this->pdo->setTicket($ticket);
318                     $this->data = json_encode( $newTicket);
319         }
320          else{
321              $this->data="";
322              $this->codeRetour=400;
323          }
324     }
```

La méthode *verifierVisiteur* de la classe PDO va vérifier le login, le mot de passe ainsi que la validité du timespan courant :

```
75          public function verifierVisiteur($login, $mdpHache){
76                  $req = "select * from visiteur where login = :login";
77                  $stm = self::$monPdo->prepare($req);
78                  $stm->bindParam(':login', $login);
79                  $stm->execute();
80                  $laLigne = $stm->fetch();
81                  if(count($laLigne)>1){
82                          $mdp = $laLigne['mdp'];
83                          $ticket = $laLigne['ticket'];
84                          $dernierTimespan = $laLigne['timespan'];
85                          $timespanAutorise = time() - 600;
86                          $verif = sha1($ticket . $mdp);
87                          if($mdpHache == $verif &&  $dernierTimespan > $timespanAutorise){
88                                  return $ticket;
89                          }
90                  }
91                  return NULL;
```

- Ligne 85, on calcule le timespan autorisé (600 correspond à 10 minutes)
- Ligne 86, on hache le tickect en base + le mot de passe en base
- Ligne 87, si deux hachages sont identiques et si le timespan courant est correct on retourne le ticket.

Remarque
On peut penser que le test sur le timespan est superflu puisque les deux appels REST sont joués sur le même événement click du bouton ; néanmoins, il est nécessaire dans l'hypothèse où le second appel REST est joué seul dans un scénario de hacking. N'oublions pas qu'un appel REST peut être lancé sans restriction sur une URL ; c'est donc au serveur de prévoir les scénarios malfaisants.

2) *Modification du code, côté consommateur du service REST (l'application Angular)*

La classe *DataService* doit être modifiée puisqu'à la connexion deux requêtes seront jouées :

```
14
15      public login(loginIn : string) : Observable<string>{
16          let url :string = this.urlDomaine + "/login?login=" + loginIn;
17          let req = this.http
18                      .get(url)
19                      .map((r: Response)=>r.json());
20          return req;
21      }
22      public connexion( loginIn : string, mdpHache : string ) : Observable<string> {
23          let url :string = this.urlDomaine + "/connexion?login=" + loginIn + "&mdp=" + mdpHache;
24          let req = this.http
25                      .get(url)
26                      .map((r: Response)=>r.json());
27          return req;
28      }
```

L'une pour lancer le premier appel REST sur le seul login (méthode login), l'autre pour lancer le second appel REST avec le login et le mot de passe (haché).

C'est dans le composant de connexion que l'on doit s'abonner à ces flux :

```
23
24    constructor(private router : Router,private dataService : DataService, private sha1Service : Sha1Service){}
25
26    valider():void{
27            this.dataService.login(this.login)
28                                .subscribe(
29                                          (data)=>{this.dataService.ticket = data;
30                                                   this.connexion()}
31                                         ,(error)=>{this.estCache = false;
32                                                  this.lblMessage ="erreur de login";
33                                                  }
34                                    );
35    }
36    private  connexion(){
37             let mdpHache : string = this.sha1Service.hash(this.dataService.ticket + this.mdp);
38             this.dataService.connexion(this.login, mdpHache)
39                             .subscribe(
40                                          (data)=>{this.dataService.ticket = data;
41                                                  this.router.navigate(['accueil']);}
42                                         ,(error)=>{this.estCache = false;
43                                                  this.lblMessage += " erreur de mot de passe";}
44                                                  );
45
46      }
47  }
48
```

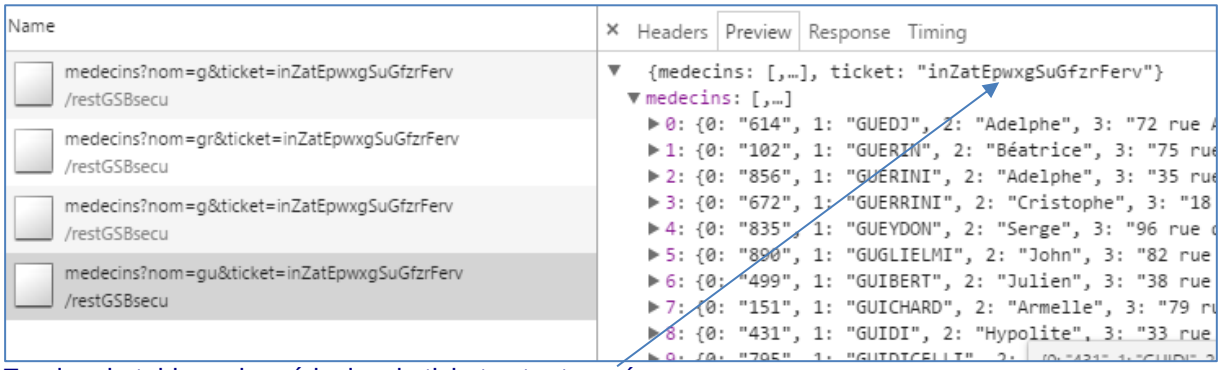
- Ligne 24, un **nouveau service** est injecté permettant de hacher (sha1) le mot de passe. Il n'est pas nécessaire de commenter cette classe de hachage : elle fait le travail…
- La méthode *valider* (sur le click) s'abonne au flux retourné par la méthode login ; dans l'hypothèse où des données sont retournées (ligne 29), on lance le code de la méthode connexion (ligne 30) ; en effet, par soucis de lisibilité, ce code a été déporté dans une méthode spécifique (ligne 36).
- Ligne 29, le ticket est récupéré à partir du serveur.
- Ligne 37, le mot de passe est haché avec un *sel* -le ticket-.

Mise à jour de la gestion des médecins

Nous allons commenter le mécanisme de sécurisation pour la partie concernant la gestion des médecins.

1) Côté serveur

Pour la recherche du médecin, l'appel REST contient le ticket courant :



En plus du tableau de médecins, le ticket est retourné

La méthode associée à cet appel est modifiée :

```
317     private function getLesMedecins($args){
318
319         $ticket = $args['ticket'];
320         if($this->pdo->estTicketValide($ticket)){
321             $nom = $args['nom'];
322             $lesLignes = array();
323             $lesMedecins = $this->pdo->getLesMedecins($nom);
324             $lesLignes['medecins'] =  $lesMedecins;
325             $lesLignes['ticket'] = $ticket;
326             $this->data = $this->encoderReponse( $lesLignes);
327         }
328         else{
329             $this->data="";
330             $this->codeRetour=400;
331         }
```

- Ligne 320, un test est effectué pour déterminer la validité du ticket ; c'est la méthode *estTicketValide* qui fait ce travail
- Les lignes 322 à 326, déclarent, construisent et retournent ce tableau

```
204         public function estTicketValide($ticket){
205                 $req = "select * from visiteur where ticket = :ticket";
206                 $stm = self::$monPdo->prepare($req);
207                 $stm->bindParam(':ticket', $ticket);
208                 $stm->execute();
209                 $laLigne = $stm->fetch();
210                 $timespanAutorise = time() - 600;
211                 $dernierTimespan = $laLigne['timespan'];
212                 $ret = 0;
213                 if(count($laLigne)>1 && $dernierTimespan > $timespanAutorise)
214                         $ret = 1;
215                 return $ret;
216         }
```

La ligne 213 effectue un test sur l'existence du ticket en base et sur la valeur de son *timespan*.

De même pour récupérer les rapports du médecin :

```
352         private function getLesRapports($args){
353
354             $ticket = $args['ticket'];
355              if($this->pdo->estTicketValide($ticket)){
356                     $idMedecin = $args['idMedecin'];
357                     $lesLignes = array();
358                     $lesRapports = $this->pdo->getLesRapports($idMedecin);
359                     error_log(print_r( $lesRapports,true),3,"log.txt");
360                     $newTicket = $this->pdo->setTicket($ticket);
361                     $lesLignes['rapports'] = $lesRapports;
362                     $lesLignes['ticket'] = $newTicket;
363                     $this->data = $this->encoderReponse( $lesLignes);
364              }
365              else{
366                     $this->data ="Erreur";
367                     $this->codeRetour=400;
368
369              }
```

- Ligne 359, un moyen bien pratique pour loguer des variables dans un fichier !
- Ligne 360, un nouveau ticket est généré
- Lignes 361 et 382, on construit un tableau contenant les données et le nouveau ticket que l'on va retourner à l'application.

2) Côté application

Regardons, dans la classe *DataService*, la modification concernant la récupération des rapports :

```
39      public chargerRapports(idMedecin : Number ): Observable<string[]>{
40         let url : string =  this.urlDomaine + "/rapports?idMedecin=" + idMedecin + "&ticket=" + this.ticket;
41         let req = this.http
42                     .get(url)
43                     .map((r: Response)=>r.json());
44          return req;
45      }
```

Le ticket est ajouté à la requête.

Dans la classe *MedecinsComponent* :

```
42      }
43      derniersRapports() : void{
44           this.afficherRapports = true;
45           this.afficherMedecin = false;
46           this.dataService.chargerRapports(this.medecin.id)
47                                    .subscribe(
48                                         (data)=>{this.lesRapports = data['rapports'];
49                                         this.nomMedecin="";
50                                         this.dataService.ticket = data['ticket']
51                                            }
52                                         ,(error)=>{this.router.navigate(['']);}
53                                                 );
54      }
```

La modification porte sur la récupération des rapports et du ticket ainsi que sur la gestion d'une erreur (ligne 52) qui renvoie au formulaire de connexion ; l'erreur est envoyée par le serveur (erreur 404).

Les modifications des appels REST auront toujours cette même forme : récupération du ticket, envoi vers le formulaire de connexion si une erreur est retournée. Ainsi l'application est toujours à même de lancer une requête REST avec un ticket à jour.

Travail à faire

En vous inspirant de ce qui est présenté (et présent dans l'application V5.0), réaliser les modifications nécessaires dans l'application Angular2 pour gérer la sécurité des appels REST pour la gestion des rapports (mise à jour et ajout).

Vous trouverez ici le corrigé : https://github.com/patricegrand/GSBAngular2_V5.1

Annexe 1 : quelques rappels indispensables

Le chiffrement ou cryptage

C'est le traitement qui consiste à transformer une information à l'aide d'algorithme(s) utilisant une clé (chaine de caractères de longueur limitée) ; si la même clé est utilisée pour chiffrer et déchiffrer, on parle de chiffrement symétrique, si ce n'est pas le cas on parle de chiffrement asymétrique. Deux aspects sont à bien intégrer :

- Le chiffrement ne diminue pas la taille du document source
- **Un document chiffré peut être déchiffré**, c'est même un de ses intérêts (à condition que l'on connaisse l'algorithme et la clé bien sûr).

Le mécanisme est relativement sûr et est utilisé dans de nombreux domaines (téléphonie chiffrée, audiovisuel, protection des données informatiques...)

Le hachage

Il s'agit de tout autre chose, même si l'on se trouve dans le même domaine -celui de la sécurité-. Hacher un document, c'est le chiffrer mais avec le double objectif d'en réduire sa taille et de rendre sa « source » non prédictible ; on ne peut pas extraire la source d'un document haché, on ne peut que vérifier qu'un document X haché est identique à un document Y haché. Il existe différentes fonctions de hachage MD5, SHA1 (2 ou 3) ou Sha-256 qui se distinguent notamment par la taille du haché rendu.

Le support utilise SHA1, qui est supporté par PHP.

Toute donnée hachée (inférieure à 2^64 bits soit -très- approximativement 100 milliards de caractères) avec SHA1 peut être réduite à 40 caractères.

Il s'en suit tout naturellement que deux sources peuvent avoir le même haché, on dit la même *empreinte* ; ceci s'exprime par ce que l'on appelle le risque de collision. Concernant SHA1, ce risque de collision est de 2^63 ; ou dit autrement : il faudrait générer aléatoirement un million de milliards de documents pour que l'on retrouve la même empreinte (la même valeur hachée). Ce risque de collision est encore jugé trop important aujourd'hui, ce qui fait que la fonction de hachage recommandée est SHA-256.

Il existe de nombreux sites qui décrivent les différentes techniques pour « hacker » des données hachées ; ces techniques sont basées principalement sur des *rainbow tables*, « dictionnaires » de couples donnée/donnée hachée qu'il est très facile à réaliser par ailleurs puisque l'algorithme de hachage est totalement public (notre support utilise ainsi une fonction de hachage SHA1 faite à la main -pas la mienne…-). Pour terminer sur ce passage sur le hacking du hachage SHA1, le dictionnaire évoqué plus haut est nettement réduit si l'on s'attaque à un mot de passe dont le nombre de caractères est réduit à un intervalle 4 caractères – 10 caractères !! C'est pour cela d'ailleurs que le hachage n'est pas en soi une sécurité mais s'accompagne d'autres techniques dont l'une est très souvent utilisée, le *grain de sel* (salt).

Il s'agit, dans ce dernier cas, d'ajouter à la donnée à protéger une chaîne de caractères relativement longue en début et/ou en fin de donnée :
mdp = 'toto' ;
avec grain de sel : 'zZ ;4 !!Ftg0'.'toto'.'zZ ;4 !!Ftg0'
que l'on va hacher ; la présence de cette suite dans les *rainbow tables* est moins probable.