JSON Web Tokens

Douae MAHROUG (FA)
16/11/2017

Résumé

Un **token** (un jeton) est simplement une **String** qui emporte une structure. Un **JWT** (JSON Web Token) est un standard ouvert (RFC 7519) pour échanger de l'information de manière sécurisée via un jeton signé. Par exemple un serveur pourrait émettre un jeton possédant l'affirmation "utilisateur identifié en tant qu'administrateur" et le fournir au client. Le client pourrait alors vérifier le jeton pour prouver que l'utilisateur est identifié en tant qu'administrateur.

Dans ce TP, nous verrons la structure des JSON Web Tokens, et comment utiliser ces derniers pour sécuriser notre code et nos applications.

Pré-requis

* Avoir des notions d'Android

* Connaître la structure d'une données JSON

* Avoir quelques notions de la structure REST

Code source

Code source **initial** disponible à :
https://github.com/mahrougd/JsonWebTokens_initial.git

Code source **final** disponible à :
https://github.com/mahrougd/JsonWebTokens_final.git

Explications du TP

Pour commencer ce TP, nous allons expérimenter le fonctionnement d'un JWT, mais avant il faut savoir que ce dernier se compose de trois parties séparées par un point : **part1.part2.part3**

part1 : Le header encodé en base64
Le header contient un objet indiquant l'algorithme utilisé pour "hasher" le contenu. Par exemple :

{ "alg": "HS256", "typ": "JWT" } ⇒ **eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9**

part2 : Le contenu encodé en base64 (JWT claims)

Le contenu est un simple objet avec les informations à échanger entre le client et le serveur. On peut y placer librement des champs personnalisés et propres à nos besoins. Par e*xemple* :

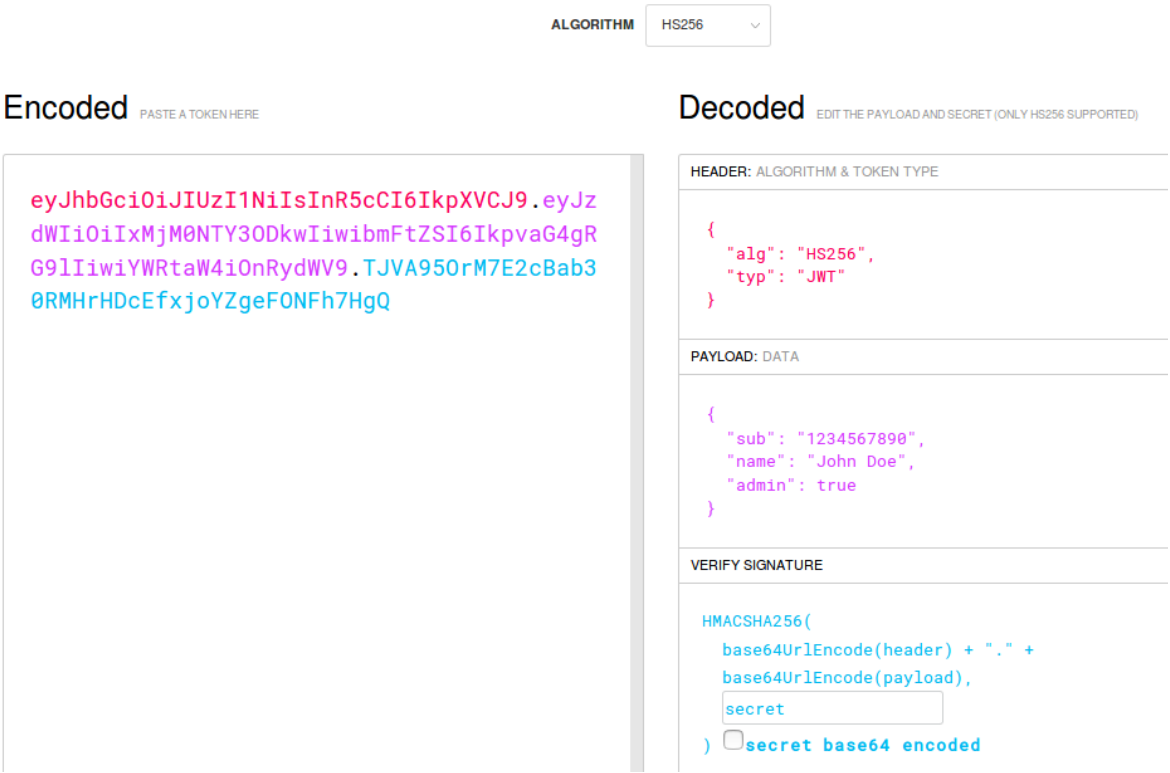
{"sub": "123","name": "John","admin": true} ⇒
eyJzdWIiOiIxMjMiLCJuYW1lIjoiSm9obiIsImFkbWluIjp0cnVlfQ

part3 : La signature

Pour générer la signature, on concatène le header et le contenu puis on encode avec l'algorithme défini dans le header. Par e*xemple* :

HMACSHA256(
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 + « . » +
eyJzdWIiOiIxMjMiLCJuYW1lIjoiSm9obiIsImFkbWluIjp0cnVlfQ,
« secret key »)

Vue globale :



Passons maintenant à l'expérimentation des JWT sous Android studio :

Pour cela on ajoute la dépendance suivante dans Gradle :

```
dependencies {
    compile 'io.jsonwebtoken:jjwt:0.9.0'
}
```

Ensuite dans MainActivity :

```
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.impl.crypto.MacProvider;
import java.security.Key;


Key key = MacProvider.generateKey();
String compactJws = Jwts.builder()
        .setSubject("tpjws")
        .signWith(SignatureAlgorithm.HS512, key)
        .compact();
String msgTxt =  Jwts.parser()
     .setSigningKey(key)
     .parseClaimsJws(compactJws)
     .getBody().getSubject();
```

Maintenant qu'on connaît le fonctionnement d'un jwt nous allons apprendre à les utiliser dans une API REST, pour cela nous allons utiliser Retrofit, qui est une librairie permettant de réaliser des appels à des webservices REST sur Android.

Etape 1 : Appeler un webservice REST

Nous allons commencer par importer la librairie Retrofit dans notre projet en utilisant Gradle. Pour cela il suffit d'y ajouter la ligne suivante :

```
dependencies {
    compile 'com.squareup.retrofit:retrofit:1.9.0'
}
```

Nous allons interagir avec l'API de github :

Retrofit se base sur un fichier de description de notre API REST :

```
import java.util.List;

import retrofit.http.GET;
import retrofit.http.Path;
import retrofit.http.Query;

public interface GithubAPI {

    public static final String ENDPOINT = "https://api.github.com";

   @GET("/users/{user}/repos")
   List<Depot> listDepots(@Path("user") String user);

   @GET("/search/repositories")
   List<Depot> searchDepot(@Query("q") String query);
}
```

Par la suite nous allons définir l'objet **Depot.**

Vous pouvez consulter l'url suivante : **https://api.github.com/users/mahrougd/repos** pour voir le retour du webservice. Dans ce TP, nous allons utiliser les attributs id, name, full_name et html_url (du retour de notre webservice), ce qui donne l'objet suivant:

```
public class Depot {

  private int id;
  private String name;
  private String full_name;
  private String html_url;
}
```

Ensuite nous allons fournir notre interface à l'objet nommé **RestAdapter,** qui nous retournera une implémentation de notre webservice :

```
GithubAPI githubapi = new RestAdapter.Builder()

  .setEndpoint(GithubAPI.ENDPOINT)
  .build()
  .create(GithubAPI.class);
```

Etape 2 : Appel synchrone

Les appels synchrone permettent d'effectuer une requête et aussi d'obtenir le retour du webservice directement depuis l'appel de la méthode de notre **GithubAPI.**

```
List<Depot> depotList = githubapi.listDepot(user);
```

```
public class MainActivity extends Activity {


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        new ListDepotsTask().execute("mahrougd");
    }

    public void afficherDepots(List<Depot> depots) {
        Toast.makeText(this,"nombre de dépots : "+depots
.size(),Toast.LENGTH_SHORT).show();
    }

    class ListDepotsTask extends AsyncTask<String,Void,List<Depot>>{

        @Override
```

```
        protected List<Depot> doInBackground(String params) {
            GithubAPI githubapi = new RestAdapter.Builder()
                    .setEndpoint(GithubAPI.ENDPOINT)
                    .build()
                    .create(GithubAPI.class);

            String user = params[0];
            List<Depot> depotsList = githubapi.listDepots(user);

            return depotsList;
        }

        @Override
        protected void onPostExecute(List<Depot> depots) {
            super.onPostExecute(depots);
            afficherDepots(depots);
        }
    }

}
```

Etape 3 : Appel asynchrone

L'appel asynchrone peux être effectué depuis le thread principal,
Le retour du webservice s'effectue par un callback, un objet dont les méthodes seront appelées suite à la réception de données du webservice :

```
public class MainActivity extends ActionBarActivity {


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        GithubAPI githubapi = new RestAdapter.Builder()
                .setEndpoint(GithubAPI.ENDPOINT)
                .build()
                .create(GithubAPI.class);

        githubapi.listDepotsAsync("mahrougd",new Callback<List<Depot>>() {
            @Override
            public void success(List<Depot> depots, Response response) {
                afficherDepots(depots);
```

```
        }

        @Override
        public void failure(RetrofitError error) {
        }
      });
   }

   public void afficherDepots(List<Depot> depots) {
      Toast.makeText(this,"nombre de dépots :
"+depots.size(),Toast.LENGTH_SHORT).show();
   }
}
```

Pour effectuer les appels de cette façon, il est nécessaire de modifier notre interface **GithubAPI** afin d’ ajouter les callbacks :

```
public interface GithubAPI {


   ...

   @GET("/users/{user}/repos")
   void listDepotsAsync(@Path("user") String user, Callback<List<Depot>>
callback);

   ...
}
```

Informations complémentaires

Webographie commentée:

- https://github.com/jwtk/jjwt : Définit la structure d'un JSON Web Token et indique les étapes à suivre pour son utilisation.
- https://fr.wikipedia.org/wiki/JSON_Web_Token : permet à un débutant de comprendre le fonctionnement d'un JSON Web Token
- https://github.com/square/retrofit : Le site de Retrofit qui permet de réaliser des appels à des webservices REST.