JWT

# JSON WEB TOKENS

When a user sends a request with required parameters like username and password. The application checks if username and password are valid. On validation, the application will create a token using a payload and a secret key. It will then send the token back to the user to store and send it with each request. When user sends request with this token, application verifies validity with same secret key. If the token is valid, the request is served, else the application will send an appropriate error message.

JSON Web Token (JWT) is an open standard ([RFC 7519](https://tools.ietf.org/html/rfc7519)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA**.

Although JWTs can be encrypted to also provide secrecy between parties, we will focus on signed tokens. Signed tokens can verify the integrity of the claims contained within it, while encrypted tokens hide those claims from other parties. When tokens are signed using public/private key pairs, the signature also certifies that only the party holding the private key is the one that signed it.

JWT Open standard means any one can use it .

We securely transfer info between any two bodies ..

Digitally signed .. info is verified and trusted

We can send request via url ,post, http header

Fast transmission and contain info of user .. we have token once you login in other time you just send the token in login ..

Basic structure of JWT is something like

header payload signature

* **header:** It contains token type and algorithm used to make signature. Gets encoded to base64.
* **payload:**Any custom user data like username and email.
* **signature:** Hash of encoded header, payload and a secret key.

https://jwt.io/introduction/

**$header = [**

**'typ' => 'JWT',**

**'alg' => 'HS256'**

**];**

**$payload = [**

**"country" => "india",**

**"name" => "vishal",**

**"email" => "vishal@gmail.com"**

**];**

**$signature = hash\_hmac('sha256','$header.$payload', $key, true);**

**$token = "$header.$payload.$signature";**

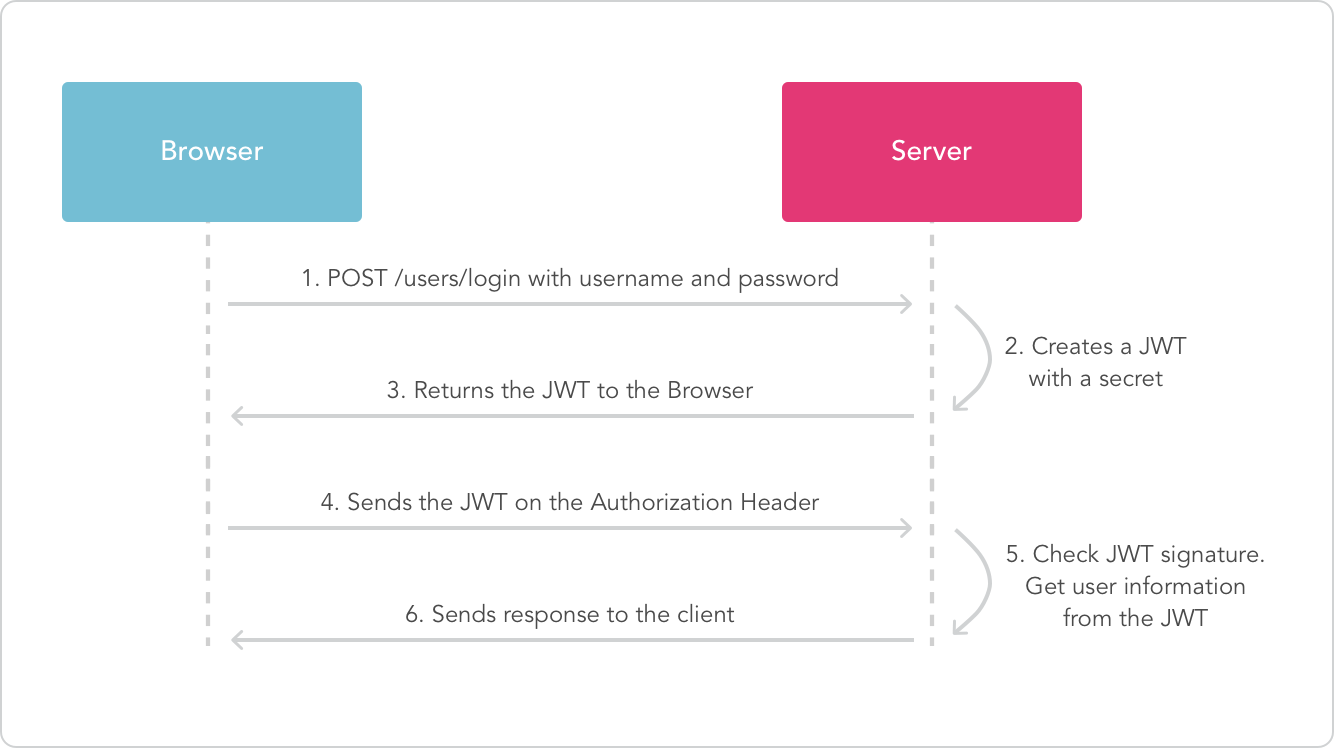
# @POST("usersInfo/Authenticate")

# Call<Response> getUserResponse(

# @Header("Token") String token,

# @Body DataUser userData

# );



# <https://jwt.io/introduction/>

WTs work in .NET, Python, Node.js, Java, PHP, Ruby, Go, JavaScript,

User login password(login,fb,google)

**server**

**browser**

**USER**  **JWT**

**Return to user**   **CREATED**

**User passes jwt while making api calls … Application server verify and process api calls**

1. **Initial login with username and password. Saved this login credentials on AccountManager.**
2. **Send login credentials to api**
3. **Api send token to client**
4. **Client save token on SharedPreferences**
5. **Interact the api with token**
6. **Question is here, if the token expire, do you resend the saved login credentials again to refresh token? OR just sending the expired token and re-generate new token? I'm thinking to do it on jwt. It is necessary to store refresh token on database**

@POST(**"login"**)  
@FormUrlEncoded  
Call<AccessToken> login(@Field(**"username"**) String username, @Field(**"password"**) String password);

**// create a class shared preference as accesstoken**  
  
**public class** TokenManager {  
  
 **private** SharedPreferences prefs;  
 **private** SharedPreferences.Editor editor;  
  
 **private static** TokenManager INSTANCE = **null**;  
  
 **private** TokenManager(SharedPreferences prefs){  
 **this**.prefs = prefs;  
 **this**.editor = prefs.edit();  
 }  
  
 **static synchronized** TokenManager getInstance(SharedPreferences prefs){  
 **if**(INSTANCE == **null**){  
 INSTANCE = **new** TokenManager(prefs);  
 }  
 **return** INSTANCE;  
 }  
  
 **public void** saveToken(AccessToken token){  
 editor.putString(**"ACCESS\_TOKEN"**, token.getAccessToken()).commit();  
 editor.putString(**"REFRESH\_TOKEN"**, token.getRefreshToken()).commit();  
 }  
  
 **public void** deleteToken(){  
 editor.remove(**"ACCESS\_TOKEN"**).commit();  
 editor.remove(**"REFRESH\_TOKEN"**).commit();  
 }  
  
 **public** AccessToken getToken(){  
 AccessToken token = **new** AccessToken();  
 token.setAccessToken(prefs.getString(**"ACCESS\_TOKEN"**, **null**));  
 token.setRefreshToken(prefs.getString(**"REFRESH\_TOKEN"**, **null**));  
 **return** token;  
 }  
  
  
  
}

**// Receive the Token from Server like //’**

**public class** AccessToken {  
  
 @Json(name = **"token\_type"**)  
 String tokenType;  
 @Json(name = **"expires\_in"**)  
 **int** expiresIn;  
 @Json(name = **"access\_token"**)  
 String accessToken;  
 @Json(name = **"refresh\_token"**)  
 String refreshToken;  
  
 **public** String getTokenType() {  
 **return** tokenType;  
 }  
  
 **public int** getExpiresIn() {  
 **return** expiresIn;  
 }  
  
 **public** String getAccessToken() {  
 **return** accessToken;  
 }  
  
 **public** String getRefreshToken() {  
 **return** refreshToken;  
 }  
  
 **public void** setTokenType(String tokenType) {  
 **this**.tokenType = tokenType;  
 }  
  
 **public void** setExpiresIn(**int** expiresIn) {  
 **this**.expiresIn = expiresIn;  
 }  
  
 **public void** setAccessToken(String accessToken) {  
 **this**.accessToken = accessToken;  
 }  
  
 **public void** setRefreshToken(String refreshToken) {  
 **this**.refreshToken = refreshToken;  
 }  
}

the API you're interacting with will send you a new token with each request before the old one expires

public String getJsonString2(String url, String jwtString) {

String jsonString = "";

// Creating HTTP client and post

HttpClient httpClient = new DefaultHttpClient();

HttpPost httpPost = new HttpPost(url);

httpPost.setHeader("Authorization", "Bearer \\{" + jwtString + "\\}");

// Making HTTP Request

try {

HttpResponse response = httpClient.execute(httpPost);

jsonString = EntityUtils.toString(response.getEntity(), HTTP.UTF\_8);

System.out.println("Http String content: " + jsonString);

} catch (ClientProtocolException e) {

// writing exception to log

e.printStackTrace();

} catch (IOException e) {

// writing exception to log

e.printStackTrace();

}

return jsonString;

}

public interface UserService {

@Headers({

"Accept: application/vnd.yourapi.v1.full+jwtstring",

"User-Agent: Your-App-Name"

})

@GET("/tasks/{task\_id}")

Call<Task> getTask(@Path("task\_id") long taskId);

}

it depends on server side services how it receives or store and get back the results

Thanks

String Token

**package** com.sessionloginviewretrofit.tokenmanager;  
  
**import** android.content.Context;  
**import** android.content.Intent;  
**import** android.content.SharedPreferences;  
  
**import** java.util.HashMap;  
  
*/\*\*  
 \* Created by lenovo on 4/10/2018.  
 \*/***public class** TokenManager {  
 *// Shared Preferences* SharedPreferences **pref**;  
  
 *// Editor for Shared preferences* SharedPreferences.Editor **editor**;  
  
 *// Context* Context **\_context**;  
  
 *// Shared pref mode* **int PRIVATE\_MODE** = 0;  
  
 *// Sharedpref file name* **private static final** String ***PREF\_NAME*** = **"mypref"**;  
  
 *// All Shared Preferences Keys* **private static final** String ***IS\_LOGIN*** = **"IsLoggedIn"**;  
  
 *// User name (make variable public to access from outside)* **public static final** String ***KEY\_NAME*** = **"name"**;  
 **public static final** String ***KEY\_ACCESSKEY*** = **"key\_name"**;  
 *// Email address (make variable public to access from outside)* **public static final** String ***KEY\_EMAIL*** = **"email"**;  
  
 *// Constructor* **public** TokenManager(Context context){  
 **this**.**\_context** = context;  
 **pref** = **\_context**.getSharedPreferences(***PREF\_NAME***, **PRIVATE\_MODE**);  
 **editor** = **pref**.edit();  
 }  
  
 */\*\*  
 \* Create login session  
 \* \*/* **public void** createLoginSession(String keyname,String name, String email){  
 *// Storing login value as TRUE* **editor**.putBoolean(***IS\_LOGIN***, **true**);  
  
 *// Storing name in pref* **editor**.putString(***KEY\_NAME***, name);  
 **editor**.putString(***KEY\_ACCESSKEY***, keyname);  
 *// Storing email in pref* **editor**.putString(***KEY\_EMAIL***, email);  
 *// commit changes* **editor**.commit();  
 }  
  
 */\*\*  
 \* Check login method wil check user login status  
 \* If false it will redirect user to login page  
 \* Else won't do anything  
 \* \*/* **public void** checkLogin(){  
 *// Check login status* **if**(!**this**.isLoggedIn()){  
 *// user is not logged in redirect him to Login Activity  
 // Intent i = new Intent(\_context, LoginActivity.class);  
 // Closing all the Activities  
 //i.addFlags(Intent.FLAG\_ACTIVITY\_CLEAR\_TOP);  
  
 // Add new Flag to start new Activity  
 //i.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);  
  
 // Staring Login Activity  
 // \_context.startActivity(i);* }  
  
 }  
  
  
  
 */\*\*  
 \* Get stored session data  
 \* \*/* **public** HashMap<String, String> getUserDetails(){  
 HashMap<String, String> user = **new** HashMap<String, String>();  
 *// user name* user.put(***KEY\_NAME***, **pref**.getString(***KEY\_NAME***, **null**));  
  
 *// user email id* user.put(***KEY\_EMAIL***, **pref**.getString(***KEY\_EMAIL***, **null**));  
  
 *// return user* **return** user;  
 }  
  
 */\*\*  
 \* Clear session details  
 \* \*/* **public void** logoutUser(){  
 *// Clearing all data from Shared Preferences* **editor**.clear();  
 **editor**.commit();  
  
 *// After logout redirect user to Loing Activity  
 // Intent i = new Intent(\_context, LoginActivity.class);  
 // Closing all the Activities  
 // i.addFlags(Intent.FLAG\_ACTIVITY\_CLEAR\_TOP);  
  
 // Add new Flag to start new Activity  
 //i.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK);  
  
 // Staring Login Activity  
 //\_context.startActivity(i);* }  
  
 */\*\*  
 \* Quick check for login  
 \* \*\*/  
 // Get Login State* **public boolean** isLoggedIn(){  
 **return pref**.getBoolean(***IS\_LOGIN***, **false**);  
 }  
}

**package** com.sessionloginviewretrofit.viewmodel;  
  
**import** android.util.Base64;  
**import** android.util.Log;  
  
**import** java.io.UnsupportedEncodingException;  
  
*/\*\*  
 \* Created by lenovo on 4/8/2018.  
 \*/***public class** JWTUtils {  
  
 **public static void** decoded(String JWTEncoded) **throws** Exception {  
 **try** {  
 String[] split = JWTEncoded.split(**"\\."**);  
 Log.*d*(**"JWT\_DECODED"**, **"Header: "** + *getJson*(split[0]));  
 Log.*d*(**"JWT\_DECODED"**, **"Body: "** + *getJson*(split[1]));  
 } **catch** (UnsupportedEncodingException e) {  
 *//Error* }  
 }  
  
 **private static** String getJson(String strEncoded) **throws** UnsupportedEncodingException{  
 **byte**[] decodedBytes = Base64.*decode*(strEncoded, Base64.***URL\_SAFE***);  
 **return new** String(decodedBytes, **"UTF-8"**);  
 }  
}

## What is Token Authentication?

The process by which an application confirms user identity is called [authentication](https://docs.stormpath.com/rest/product-guide/latest/auth_n.html). Traditionally, applications have persisted identity through session cookies which rely on session IDs stored server-side. In this structure, developers are forced to create session storage that is either unique and server-specific, or implemented as a completely separate session storage layer.

Token authentication is a more modern approach, designed solve problems server-side session IDs can’t. Using tokens in place of session IDs can lower your server load, streamline permission management, and provide better tools for supporting a distributed or cloud-based infrastructure. In this method, tokens are generated for your users after they present verifiable credentials. The initial authentication could be by  username/password credentials, API keys or even tokens from another service.

IS TOKEN REALLY SECURE

* Store your JWTs in [secure, HttpOnly cookies](https://stormpath.com/blog/jwt-the-right-way). This prevents [Cross-Site Scripting (XSS)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)) attacks.
* If you’re using cookies to transmit your JWTs, CSRF protection is super duper important! Your cookies can be used maliciously by other domains that make requests to your website without your user’s consent. If your server blindly authenticates a user, simply because they have a cookie, then you’ve got more problems than your hard drive size. You’re also allowing CSRF attacks, where other websites trigger state-changing actions on your server without your users’ consent. This is possible because the browser will always send the user’s cookies automatically, regardless of how the request was triggered. Use one of the many [CSRF Prevention](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)) measures to reduce this risk.
* Sign your tokens with a strong key that is available ONLY to the authentication service. Every time you use a token to authenticate a user, your server MUST verify that the token was signed with your secret key.
* Do not store any sensitive data in a JWT. These tokens are usually signed to protect against manipulation (not encrypted) so the data in the claims can be easily decoded and read. Encrypt your tokens if you have to put sensitive, non-opaque information in them. The secret signing key should only be accessible by the issuer and the consumer; it should not be accessible outside of these two parties.
* If you are worried about replay attacks, include a nonce (jti claim), expiration time (exp claim), and creation time (iat claim) in the claims. These are well defined in the [JWT Spec](https://tools.ietf.org/html/rfc7519).

## JJWT, JSONWebToken.io, and JWT Inspector

Stormpath supports the development of several JWT-related, open-source developer tools, including:

### JJWT

[JJWT](https://github.com/jwtk/jjwt) is an easy to use [tool for developers to create and verify JWTs in Java](https://stormpath.com/blog/jjwt-how-it-works-why). ike many libraries Stormpath supports, JJWT is completely free and open source (Apache License, Version 2.0), so everyone can see what it does and how it does it. Do not hesitate to report any issues, suggest improvements, and even submit some code!

**Username**

**USERNAME**

**USERPASS**

**LOGIN**

**SERVER**

**RETROFIT**

**Userpass Send request**

**Get JWT token**

**Send JWT**

**Header**

**Get user details**

**Step 1 : username and userpassword send to server**

**Step 2: server generate JWT Token and give response if username and password exist**

**Step 3: send back response as JWT token to client .**

**Step 4: client send it back to server for api calls so here I will get user id**