Authentication microservice

Andy

May 23, 2018

# Introduction

## Authentication challenges

Authentication microservice is the fundamentals in tweebaa ecosystem and it determines which users are allowed to access the system components. Tweebaa platform consists of web portal and mobile app, therefore, the first challenge of Authentication microservice is how to effectively authenticate users for multile applications. Also, since the platform is open and extensible, the second issue is how to integrate external third-party services.

Major issues:

* **Single sign-on (SSO)**. The user registers via Authentication microservice, entering a username and password. The app stores the password hash in the membership database. When the user logs in, the microservice verifies the password and generates a unique token for users login via different devices.
* **Social login**. The user signs in with an external service, such as Facebook, Microsoft, or Google. The app still creates an entry for the user in the membership database, but does not store any credentials. The user authenticates by signing into the external service.

## Tweebaa users

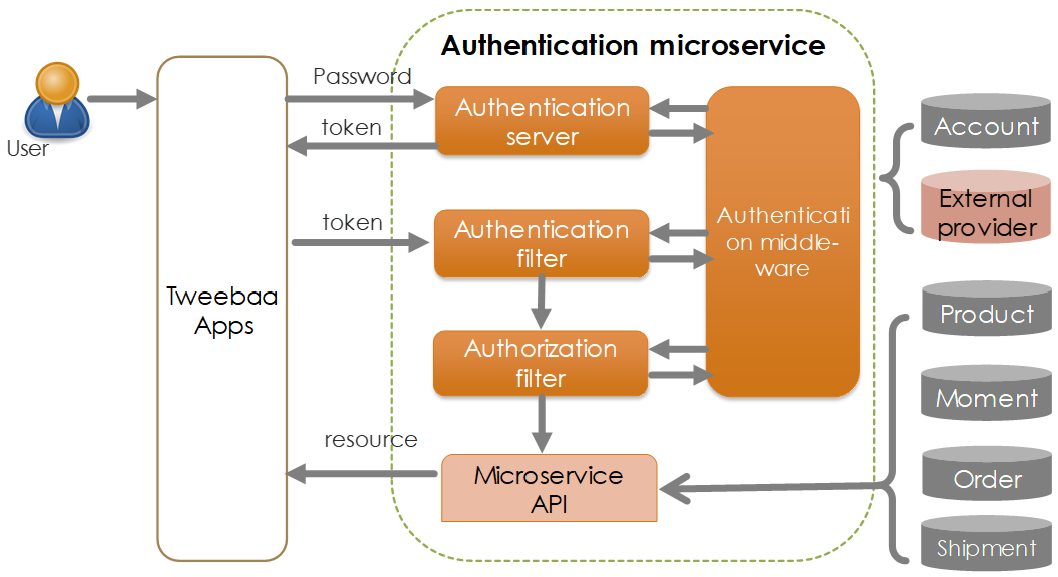
In tweebaa, users are classified into:

* Administrators - manage the platform and implement administration details
* Tycoon - the member of tweebaa ecosystem who create, share and sell products
* Vendor - the member of tweebaa ecosystem who provides products and obtain commssion
* Customer - the member restricted within an individual tycoon place who puchase products and communicates with tycoon member

The Authentication of each user includes two steps:

* Authentication - verifies the identity of the user. For example, Alice logs in with her username and password, and the server uses the password to authenticate Alice.
* Authorization - decides whether a user is allowed to perform an action. For example, Alice has permission to order a product but not create a product.

## Architecture

Here is the structure of implementing Authentication microservice: 

The major steps to access a resource is:

1. User requests access with username / password
2. Application validates credentials
3. Application provides a signed token to the client
4. Client stores that token and sends it along with every request
5. Server verifies token and responds with data

# Single sign-on (SSO)

To resolve the single sign on issue, we introduces token-based authentication technique. Token-based authentication is a security technique that authenticates the users who attempt to log in to a server, a network, or some other secure system, using a security token provided by the server. A token is a piece of data created by server, and contains information to identify a particular user and token validity. The token will contain the user’s information, as well as a special token code that user can pass to the server with every method that supports authentication, instead of passing a username and password directly.

An authentication is successful if a user can prove to a server that he or she is a valid user by passing a security token. The service validates the security token and processes the user request.After the user credentials authenticated, system issues a unique SSO token. It contains access\_token, user ID, store and expiration time. The format looks like:

{  
 "access\_token": "vVA6zsp7O0jKAZtl3DC1BmN0KwloYELbBOyhwd\_a3giUz2wiYCDouLTsmo3asubP6qxxmrFbc0qtXvhFKiMuwkcH7NnILvKqOxAHqs3\_QbQLXR\_gqn3iyWr0nKYeH0LPoIMJtSlV4bdUg7YezLGlWGANY\_DcqV\_5cLwWAOdHw07zHFYmV6-gzHKKKH8tG9D\_9QAoqhX7dXVC64u21GOUQzxghOuhMyuFkOALpeWeVVfwAqLPvm9b365t6UAsUdHT961H1AwlvOzSVC2TWOHfcIIdplwO0ff",  
 "token\_type": "bearer",  
 "expires\_in": 47303999,  
 "userName": "user-id",  
 "userGuid": "7599bf4a-6d7b-43cb-994f-7c328d96b",  
 "userId": "9521",  
 "issued": "Tue, 22 May 2018 03:11:39 GMT",  
 "expires": "Wed, 20 Nov 2019 15:11:39 GMT"  
}

After the token is validated by the service, it is used to establish security context for the client, so the service can make authorization decisions or audit activity for successive user requests.

Here is a typical code to access resources with request carries access token:

// If we already have a bearer token, set the Authorization header.  
var token = sessionStorage.getItem(tokenKey);  
var headers = {};  
if (token) {  
 headers.Authorization = 'Bearer ' + token;  
}  
   
$.ajax({  
 type: 'GET',  
 url: 'api/products/1',  
 headers: headers  
}).done(function (data) {  
 self.result(data);  
}).fail(showError);

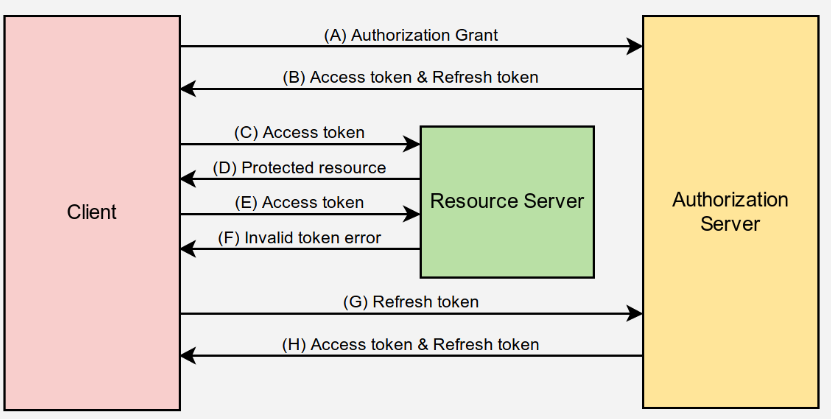
# Social login

To implement social login, we built below components:

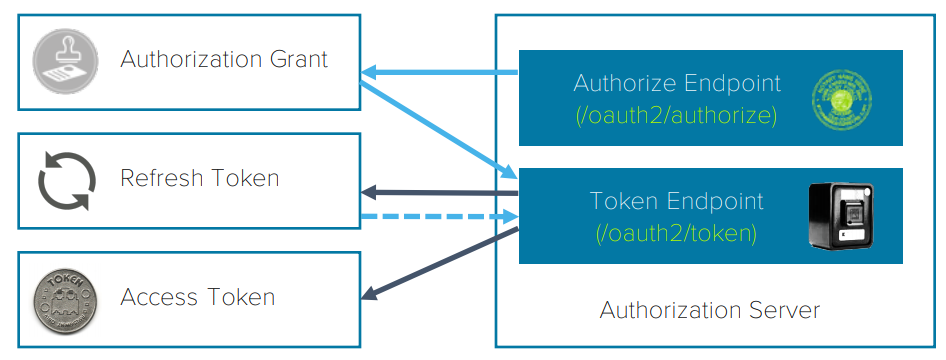
* Authorization server and Resource server based on OAuth2
* Microservice endpoints for managing user accounts
* Data models for storing user account information

## Authorization server/Resource server

Please note that authorization server is separate from resource server.



1. The authorization server asks the user for authentication (e.g. by providing the third-party username and password). Subsequently, the user can either grant or reject the client’s request. The authorization server then redirects the user to the client and passes the user’s decision to the client using a URL parameter. If the user has granted the request, the query string contains a code which the client can exchange for a security token. When doing so, the client provides authentication details to the authorization server. Mostly this is done by giving the username and password, but SecureAuth also offers a 2-Factor option via “SMS, Telephony, Email”.

* 
* The token received this way may then be used by the client to gain access to the desired resources via the resource server. The token may contain information about the user which might be used by the resource server to verify rights. Alternatively, the token may simply be a key which the resource server may exchange for user-related data when contacting the authorization server.

1. The resource server handles authenticated requests after the application has obtained an access token. It encapsulates actual business logic.

Here is the sample code to build the authentication server

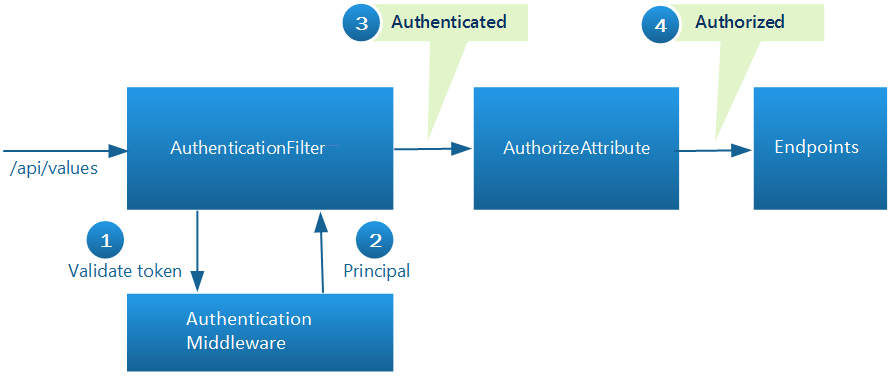
PublicClientId = "self";  
OAuthOptions = new OAuthAuthorizationServerOptions  
{  
 TokenEndpointPath = new PathString("/Token"),  
 Provider = new ApplicationOAuthProvider(PublicClientId),  
 AuthorizeEndpointPath = new PathString("/api/Account/ExternalLogin"),  
 AccessTokenExpireTimeSpan = TimeSpan.FromDays(14),  
 // Note: Remove the following line before you deploy to production:  
 AllowInsecureHttp = true  
};  
   
// Enable the application to use bearer tokens to authenticate users  
app.UseOAuthBearerTokens(OAuthOptions);

Here is the basic flow when the app wants to get a token:

1. To get an access token, the app sends a request to ~/Token.
2. The authentication middleware calls GrantResourceOwnerCredentials on the provider.
3. The provider validates the credentials and create a claims identity.
4. If that succeeds, the provider creates an authentication ticket, which is used to generate the token.

## Microservice endpoints

When the client requests a protected resource, here is what happens in the Microservice endpoint pipeline:



1. The Authentication filter calls the OAuth middleware to validate the token.
2. The middleware converts the token into a claims identity.
3. At this point, the request is authenticated but not authorized.
4. The Authorization filter examines the claims identity. If the claims authorize the user for that resource, the request is authorized.
5. If the previous steps are successful, the endpoints return the protected resource. Otherwise, the client receives a 401 (Unauthorized) error.

There are below authentication exceptions:

1. invalid\_request (HTTP 400) – The request is missing a parameter, or is otherwise malformed.
2. invalid\_token (HTTP 401) – The access token is expired, revoked, malformed, or invalid for other reasons. The client can obtain a new access token and try again.

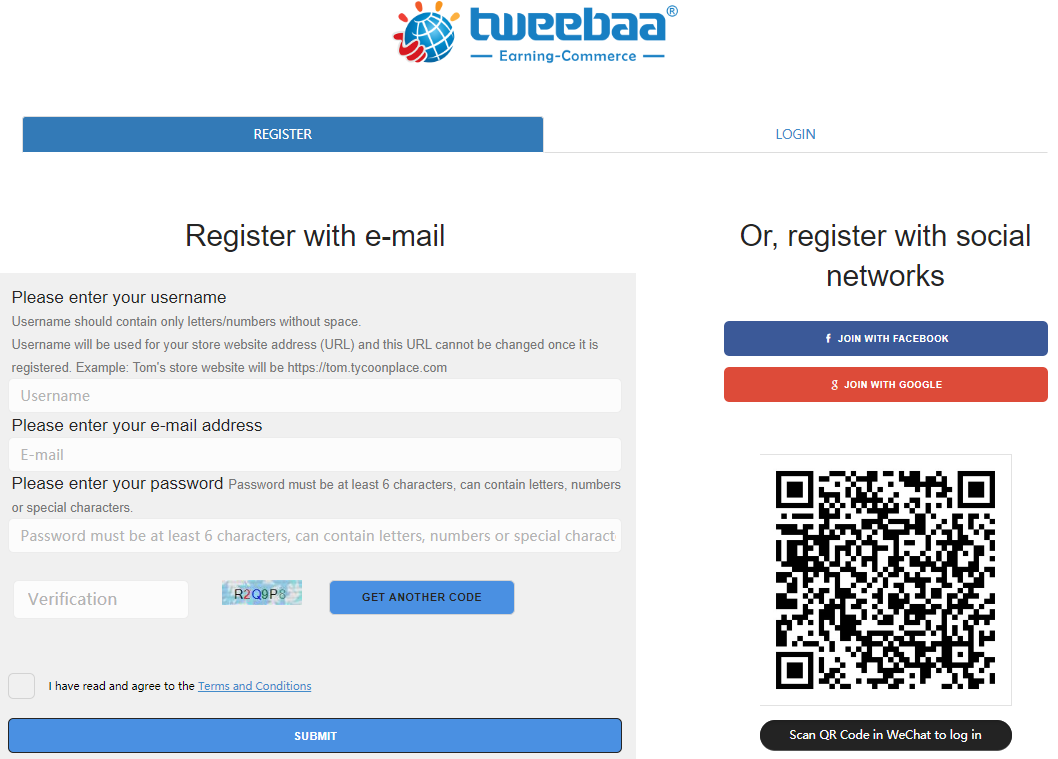
## Data models

Here are the main application classes that implement these features:

* AccountController - the endpoint for managing user accounts. The Register action is the only one that we used in this tutorial.
* ApplicationUser - the model for user accounts in the membership database.
* ApplicationUserManager - the manager performs operations on user accounts, and automatically persists changes to the database.
* ApplicationOAuthProvider - the object plugs into the Authentication middleware, and processes events raised by the middleware.

# Samples

Web authentication



Mobile app authentication 