White Paper

Securing RESTful Web Services Using Spring and OAuth 2.0

1.0 EXECUTIVE SUMMARY

While the market is hugely[1] accepting REST based architectures due to their light weight nature, there is a strong need to secure these web services from various forms of web attacks.

Since it is stateless in nature, the mechanisms of securing these services are different from standard web application where it is easily handled by session management, but in the case of REST, no session can be maintained as the calling point may or may not be a web browser.

There are several mechanisms available to secure the web service. Some of these options are: HTTP Basic, HTTP Digest, Client certificates and two legged oauth 1.0, OAuth 2.0

All of the above mechanisms can be picked along with the transport layer security using SSL.
These methods have their own pros and cons which can be easily referenced from various books and web.
This paper will primarily focus on implementing security using Spring Security for OAuth 2, an implementation of OAuth 2.0 Open Authorization standard[2].

CONTENTS

SECTION PAGE

1.0 EXECUTIVE SUMMARY............................1

2.0 BUSINESS CHALLENGES 1

3.0 OAUTH 2.0 OVERVIEW 2

4.0 SPRING SECURITY FOR OAUTH 2.0 3

5.0 APPLYING CLUSTER AWARE CACHE...6

6.0 REFERENCES 9

7.0 APPENDIX A OAUTH 2.0 OVERVIEW 9

8.0 APPENDIX B ABOUT HUGHES SYSTIQUE CORPORATION...12

1. As of 03/08/11, REST based web services constitute more than 50% of the API market available in the market. (http://www.programmableweb.com/news/3000-web-apis-trends-quickly-growing-directory/2011/03/08)
2. http://oauth.net/2

2.0 BUSINESS CHALLENGES

As against other API implementations, REST kind of API implementation has proliferated inside the enterprise and outside. At the same time, the APIs security is still in question and not mature. Because of the following reasons:

- REST architecture does not have pre-defined security methods/procedures. So developers define their own side of implementation.
- REST API is vulnerable to the same class of web attacks as standard web based applications. These include: Injection attacks, Replay attacks, Cross-site scripting, Broken authentication etc. Limited documentation/library/framework available to implement standardized OAuth 2.0 specification (see references)

3.0 OAUTH 2.0 OVERVIEW

OAuth 2.0 is an open authorization protocol specification defined by IETF OAuth WG (Working Group) which enables applications to access each other's data.
The prime focus of this protocol is to define a standard where an application, say gaming site, can access the user's data maintained by another application like facebook, google or other resource server.

The subsequent section explains the implementation of OAuth 2.0 in RESTful API using Spring Security for OAuth for Implicit Grant Type.

Those who are not familier with the OAuth roles and grant types can refer to APPENDIX A OAuth 2.0 Overview.

The following diagram gives an overview of steps involved in OAuth authentication considering a generic implicit grant scenario.

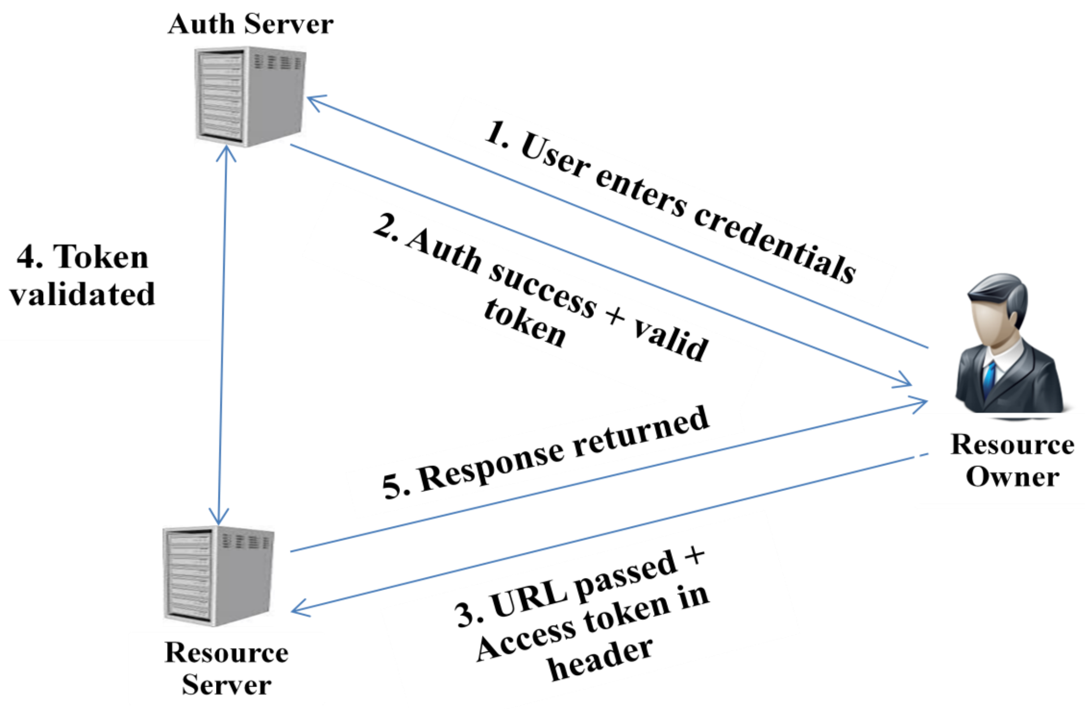
Auth Server
1. User enters credentials
2. Auth success + valid token
4. Token validated
Resource Owner
5. Response returned
3. URL passed + Access token in header
Resource Server

Figure 1: Steps involved in User Authentication

1. User enters credentials which are passed over to Authorization Server in Http Authentication header in encrypted form. The communication channel is secured with SSL.
2. Auth server authenticates the user with the credentials passed and generates a token for limited time and finally returns it in response.
3. The client application calls API to resource server, passing the token in http header or as a query string.
4. Resource server extracts the token and authorizes it with Authorization server.
5. Once the authorization is successful, a valid response is sent to the caller.

4.0 SPRING SECURITY FOR OAUTH 2.0

Spring Security provides a library (Apache License) for OAuth 2.0 framework for all 4 types of Authorization grants.

NOTE: From the implementation details perspective, this paper focuses on Implicit grant type used mostly in browser based client and mobile client applications where the user directly accesses resource server.

4.1 Spring Security Configuration

The xml configuration used for Implicit grant type is explained below.

4.1.1 Enable Spring Security configuration:

To enable spring security in your project, add this xml in the classpath and enable security filters in web.xml file as:

```
<filter>
<filter-name>springSecurityFilterChain</filter-name>
<filter-class>
      org.springframework.web.filter.DelegatingFilterProxy
</filter-class>
</filter>
<filter-mapping>
      <filter-name>springSecurityFilterChain</filter-name>
      <url-pattern>/*</url-pattern>
</filter-mapping>
```

4.1.2 Token Service

Token Service is the module which Spring Security OAuth 2.0 implementation libraries provides and it is responsible to generate and store the token as per the default token service implementation.

```
<bean                                                            id="tokenServices"
class="org.springframework.security.oauth2.provider.token.DefaultTokenService
s">
<property name="accessTokenValiditySeconds" value="86400" />
<property name="tokenStore" ref="tokenStore" />
<property name="supportRefreshToken" value="true" />
<property name="clientDetailsService" ref="clientDetails" />
</bean>
```

The tokenStore ref bean represents the token store mechanism details. In memory token store is used for good performance considering the web server has optimum memory management to store the tokens. Other alternative is to use a database to store the tokens. Below is the bean configuration used:

Bean configuration:

```
<bean id="tokenStore"
class="org.springframework.security.oauth2.provider.token.InMemoryCacheTokenS
tore">
</bean>
```

4.1.3 Authentication Manager

Authentication Manager is the module which Spring Security OAuth 2.0 implementation libraries provide and it is responsible to do the authentication based on the user credentials provided in request header [or by other means i.e using query parameters] before the framework issues a valid token to the requester.

Authentication Manager will further tie with the Authentication provider. Authentication provider will be chosen based on the underline system used to store operator details. Spring OAuth library supports various AAA systems that can be integrated and act as Authentication Provider.

More details can be found at http://docs.spring.io/spring-security/site/docs/3.1.x/apidocs/org/springframework/security/authentication/AuthenticationProvider.html

Below part of the section explains the LDAP system used as AuthenticationProvider which stores the operator details.

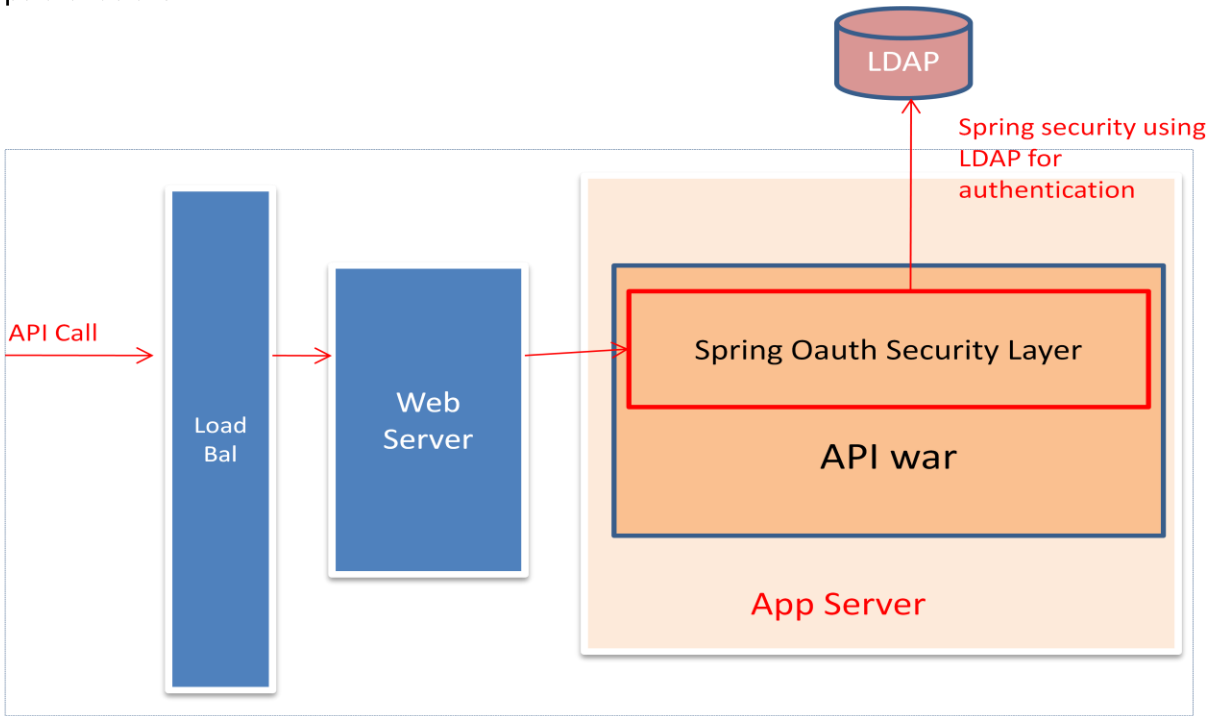
LDAP
Spring security using LDAP for authentication
API Call
Load Bal
Web Server
Spring Oauth Security Layer
API war
App Server

Figure 2: OAuth security layer access LDAP for operator access control

The LDAP configuration is given below:

```
<authentication-manager alias="authenticationManager"
        xmlns="http://www.springframework.org/schema/security">
<ldap-authentication-provider
        user-search-filter="(uid={0})"                            user-search-
base="${OperatorBindString}"
        group-search-filter="(uniqueMember={0})"                 group-search-
base="${operatorRoot}">
</ldap-authentication-provider>
</authentication-manager>

<ldap-server            xmlns="http://www.springframework.org/schema/security"
url="${providerURL}"
        manager-dn="${securityPrincipal}"    manager-password="${credentials}"
/>
```

Authentication manager will create a context object for the LDAP and verify the provided operator credentials in the request header with the details stored in the LDAP. Once authentication is successful, request will be delegated to the token service.

If Authentication is success then Token Service will first check whether a valid token is already available or not. If a valid [non-expired] token is available then that token will be returned otherwise a new token will be generated based on the token generator and added to the database and also returned as response body.
Success response body is given below.

```
  {
        "expires_in": "599",
        "token_type": "bearer",
        "access_token": "2a5249a3-b26e-4bb3-853c-6446d674ceb3"
   }
```

If authentication fails then 401 Unauthorized will be returned as response header.

4.1.4 Generate/Get access token

As per the OAuth 2.0 specification, user has to make a request for the token with certain parameter and the framework should provide a valid token to the user once authentication is done.
To access any API resource first client has to make a request for the token by providing his/her credentials in basic auth header.

Authentication Manager will do the authentication based on the credentials provided in auth header.
Below http block will receive the token request initiated by the user.

```
<http pattern="/**" create-session="stateless"
      entry-point-ref="oauthAuthenticationEntryPoint"
      xmlns="http://www.springframework.org/schema/security">
<intercept-url pattern="/**" access="IS_AUTHENTICATED_FULLY"/>
<anonymous enabled="false" />
<custom-filter ref="resourceServerFilter" before="PRE_AUTH_FILTER" />
<access-denied-handler ref="oauthAccessDeniedHandler" />
</http>
```

4.1.5 Access token request format

Below is the URL format of the token generation API:

```
https://<SERVER_URL>/HSCAPI/oauth/authorize?response_type=token&client_id=hsc
&redirect_uri= https://< SERVER_URL>/HSCAPI/tokenService/accessToken
```

Request Header:
Authorization: <base64 encoded <username:password>>

As per the OAuth 2.0 specification token request must be initiated along with these parameters. Details are given below:

response_type: It tells that this request represents the token request.
client_id: It represents the 3rd party user if token request is initiated by that. In our case there are no 3rd party users who need access to the application on behalf of operator.
redirect_uri: The URL where request will be redirected with token and other details if authentication is success.

4.2 API authorization based on user details stored in LDAP

On successful authentication via OAuth2AuthenticationProcessingFilter (authentication filter), the API Authorization layer is invoked. For customized authorization implementation, Authorization server is implemented and called from AuthenticationProcessingFilter by extending the class.

4.2.1 Authorization Server

This module is responsible to approve the token request initiated by the client. The decision would be approved by user approval handler after verifying client credentials provided in the request.

```
<oauth:authorization-server
        client-details-service-ref="clientDetails"            token-services-
ref="tokenServices"
        user-approval-handler-ref="userApprovalHandler">
<oauth:implicit />
<oauth:refresh-token />
</oauth:authorization-server>

<oauth:client-details-service id="clientDetails">
<oauth:client client-id="hsc" resource-ids="hscapi"
            authorized-grant-types="implicit"
            authorities="ROLE_CLIENT" scope="read,write" secret="secret"
            redirect-uri="${redirect-uri}" />
</oauth:client-details-service>
```

4.2.2 User Approval Handler

This handler is used to approve the token request initiated by client. In case of implicit grant type it has functionality of automatic approve for a white list of client configured in the spring-security.xml file.

```
<bean id="userApprovalHandler"
        class="oauth2.handlers.HSCAPIUserApprovalHandler">
<property name="autoApproveClients">
<set>
<value>hsc</value>
</set>
</property>
<property name="tokenServices" ref="tokenServices" />
</bean>
```

The authorization server, checks for the access of the URI and http method for the user (API caller).
If user is not authorized 401-Unauthorized status is set as response and authentication object is not set to the security context.

5.0 APPLYING CLUSTER AWARE CACHE

The OAuth token management can be done using some persisted mechanism or it can be maintained as part of process memory.

Application of memory cache for OAuth short lived token management can be used for optimal performance. And if the application is required to run in a clustered environment, a cluster aware cache mechanism is the way to go.

Rest of the section explains the usage of JBoss cache for token management.

Below is the configuration used to enable jboss caching, if JBoss AS 5.1 is the app server:

Path:	$JBOSS_HOME/.../deploy/cluster/jboss-cache-manager.sar/META-INF/jboss-cache-manager-jboss-beans.xml

```
<entry><key>api-token-store-cache</key>
      <value>
            <bean                              name="ApiTokenStoreCacheConfig"
class="org.jboss.cache.config.Configuration">

         <!-- Provides batching functionality for caches that don't want to
interact with regular JTA Transactions -->
         <property
name="transactionManagerLookupClass">org.jboss.cache.transaction.BatchModeTra
nsactionManagerLookup</property>

         <!-- Name of cluster. Needs to be the same for all members -->
         <!--property
name="clusterName">${jboss.partition.name:DefaultPartition}-
SessionCache</property-->
         <!-- Use a UDP (multicast) based stack. Need JGroups flow control
(FC)
                            because we are using asynchronous replication. --
>
         <property
name="multiplexerStack">${jboss.default.jgroups.stack:udp}</property>
                  <property name="fetchInMemoryState">true</property>

         <property name="nodeLockingScheme">PESSIMISTIC</property>
                  <property name="isolationLevel">REPEATABLE_READ</property>
                           <property name="useLockStriping">false</property>
                                    <property
name="cacheMode">REPL_ASYNC</property>

         <!-- Number of milliseconds to wait until all responses for a
                            synchronous call have been received. Make this
longer
                                          than lockAcquisitionTimeout.-->
         <property name="syncReplTimeout">17500</property>
                  <!-- Max number of milliseconds to wait for a lock
acquisition -->
         <property name="lockAcquisitionTimeout">15000</property>
                  <!-- The max amount of time (in milliseconds) we wait until
the
                                 state (ie. the contents of the cache) are
retrieved from
                                          existing members at startup. -->
         <property name="stateRetrievalTimeout">60000</property>

         <!-- Not needed for a web session cache that doesn't use FIELD -->
         <property name="useRegionBasedMarshalling">true</property>
                  <!-- Must match the value of "useRegionBasedMarshalling" --
>
         <property name="inactiveOnStartup">true</property>
         <!-- Disable asynchronous RPC marshalling/sending -->
         <property name="serializationExecutorPoolSize">0</property>
                  <!-- We have no asynchronous notification listeners -->
         <property name="listenerAsyncPoolSize">0</property>

      </bean>
    </value>
</entry>
```

More details of the configuration can be found **here** http://docs.jboss.org/jbossclustering/cluster_guide/5.1/html/jbosscache.chapt.html

5.1.1 Cache Manager Access and object storage:

First retrieve cache manager and start the cache:

```
CacheManagerLocator locator = CacheManagerLocator.getCacheManagerLocator();
//This configuration is loaded from file
        //$JBOSS_HOME/.../deploy/cluster/jboss-cache-manager.sar/META-
INF/jboss-cache-manager-jboss-beans.xml
cacheManager                                                            =
(org.jboss.ha.cachemanager.CacheManager)locator.getCacheManager(null);
        cache = cacheManager.getCache("api-token-store-cache", true);
        cache.start();
        cache.addCacheListener(new TokenListener());

Get the root node:
Node<String, Object> rootNode = cache.getRoot();
```

Define fully qualified names for cache collections nodes:

```
//Define Fully-qualified names for the token store collections
Fqn accessTokenStoreFqn = Fqn.fromString("/accessTokenStore");
Fqn auth2AccessTSFqn = Fqn.fromString("/authenticationToAccessTokenStore");
Fqn                  userNameToAccessTokenStoreFqn                      =
Fqn.fromString("/userNameToAccessTokenStore");
Fqn                  clientIdToAccessTokenStoreFqn                      =
Fqn.fromString("/clientIdToAccessTokenStore");
Fqn authenticationStoreFqn = Fqn.fromString("/authenticationStore");
Fqn expiryMapFqn = Fqn.fromString("/expiryMap");
Fqn flushCounterFqn = Fqn.fromString("/flushCounter");
```

Add token store collections to the cache at respective nodes defined by Fqns (Fully Qualified Names):

```
Node<String, Object> tsNode = rootNode.addChild(accessTokenStoreFqn);
tsNode.put("accessTokenStore", accessTokenStore);
```

Note: If you are using custom class loading mechanism, you might need to define cache regions and activate them:

```
Region cacheRegion = cache.getRegion(rootFqn, true);

cacheRegion.registerContextClassLoader(this.getClass().getClassLoader());
cacheRegion.activate();
cacheRegion.setActive(true);
```

6.0 REFERENCES

http://oauth.net/2

http://tools.ietf.org/html/rfc6749

http://tutorials.jenkov.com/oauth2/index.html

http://projects.spring.io/spring-security-oauth/

https://github.com/spring-projects/spring-security-oauth/wiki/tutorial

http://www.programmableweb.com/news/3000-web-apis-trends-quickly-growing-directory/2011/03/08

http://docs.jboss.org/jbossclustering/cluster_guide/5.1/html/jbosscache.chapt.html

7.0 APPENDIX A OAUTH 2.0 OVERVIEW

OAuth 2.0 defines following entities, a brief description is also provided.

Roles

Resource Owner: A Person or application that owns the data to be shared.
Resource Server: Server where the owner resources are stored. E.g Google, Facebook resource server
Client Application: Application requesting the access to the user data to the resource server
Authorization Server: Server authorizing the client app to access the resources of the resource server

Client Types

There are two types of client defined by OAuth 2.0 specification.

Confidential: This type keeps the user credentials confidential to the world. E.g an application where only the super or admin user can access the auth server to get access to the client credentials.
Public: Such applications are not capable to keep a client password confidential.

Other Entities

ClientID: It is the ID shared by the Authorization Server to the client application at the time of client application registration to the Authorization server. This is typically a onetime process of registration. No user's data is shared to the client app in this process.

Client Secret: It is the client application secret given by the authorization server during client application registration.

Redirect URI: During registration, the client also registers a redirect URI. This URI is used by the Authorization server to redirect the page to given URI when resource owner allows client application to access its resources.

Authorization Grant

There are four types of grants as mentioned in the OAuth 2.0 specification (see references)

- **Authorization Code:**

This type of grant is best suited for social networking sites like facebook, twitter, google giving access to user data to other applications (client applications) on user permission. The resource owner accesses the client application. The client application tells the user to login to the client application via an authorization server.

On user login request at client application site, the user is directed to the authorization server by the client application. The client application sends its client ID with the request. The Auth server knows which client is trying to request for user's protected data.

After successful login, the user is asked if he wants to grant access to its resources to the client application.

On approval, the page is redirected to the redirect URI of the client. The auth server also sends a unique authorization code.

The client application then sends the auth code, its client ID and client secret to the auth server. The auth server sends back the access token for limited time duration.

The client application can then use this access token to request resources from the resource server.

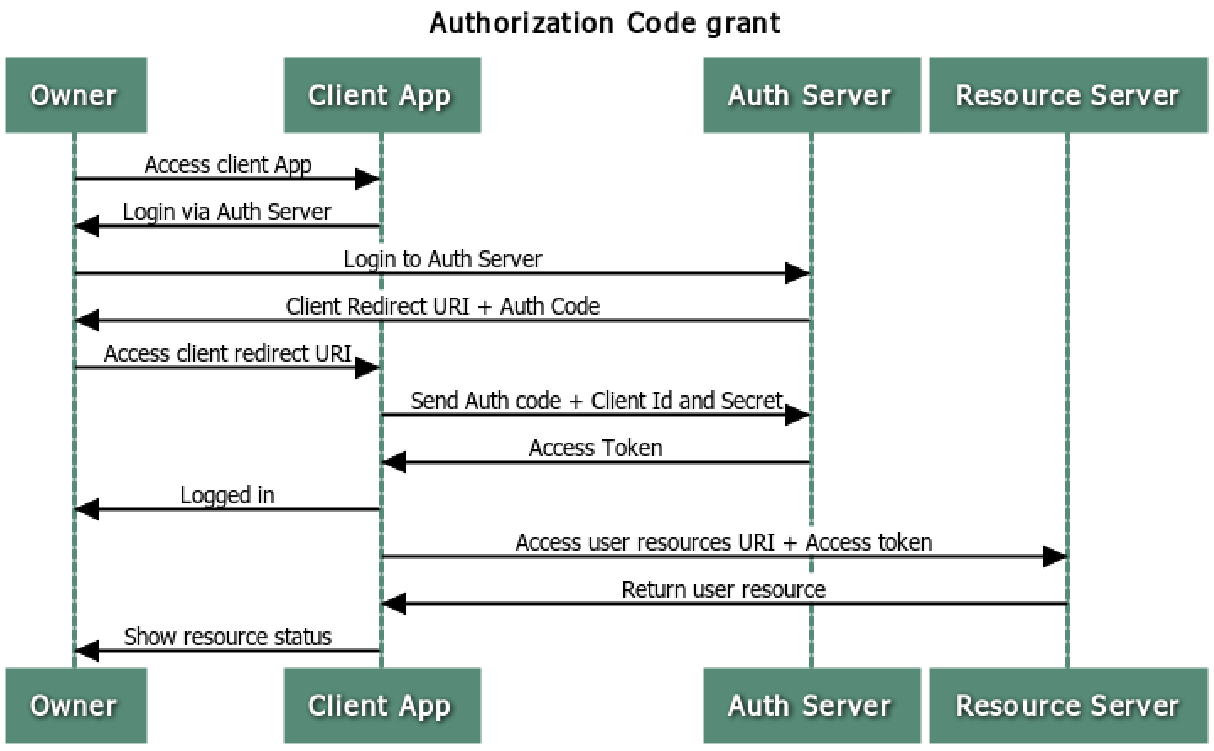

Figure 3: Authorization Code Grant Type Work Flow

- **Implicit:**

This type of grant is mostly used in browser based applications where user and client app represents one entity or native client application. It is similar to authorization code grant type except that fact that the access token is returned as soon as the user finished authorization along with the redirect URI.

In this case since the token is returned to the user, it is prone to being hacked. It is thus strongly recommended to use TLS for the implementation.

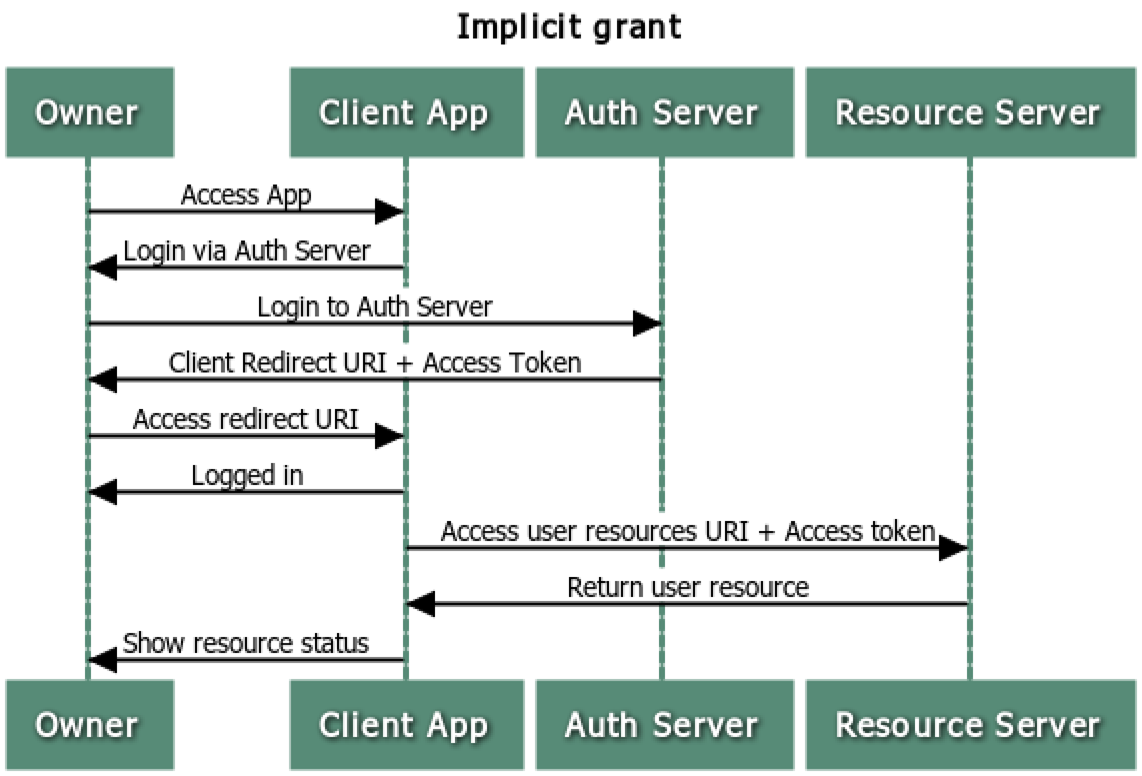

Figure 4: Implicit Grant Type Work Flow

- **Resource Owner Password Credentials**

  In resource owner password credentials authorization grant, the owner gives the client application access to its credentials. The client application can then use the user name and password to access resources.

  This type of grant is not secure.

- **Client Credentials**

  The client application needs to access resources or call functions in the resource server, which are not related to a specific resource owner.

8.0 APPENDIX B: ABOUT HUGHES SYSTIQUE CORPORATION

HUGHES Systique Corporation (HSC), part of the HUGHES group of companies, is a leading Consulting and Software company focused on Communications and Automotive Telematics. HSC is headquartered in Rockville, Maryland USA with its development centre in Gurgaon, India.

SERVICES OFFERED:

Technology Consulting & Architecture: Leverage extensive knowledge and experience of our domain experts to define product requirements, validate technology plans, and provide network level consulting services and deployment of several successful products from conceptualization to market delivery.

Development & Maintenance Services: We can help you design, develop and maintain software for diverse areas in the communication industry. We have a well-defined software development process, comprising of complete SDLC from requirement analysis to the deployment and post production support.

Testing : We have extensive experience in testing methodologies and processes and offer Performance testing (with bench marking metrics), Protocol testing, Conformance testing, Stress testing, White-box and black-box testing, Regression testing and Interoperability testing to our clients

System Integration : As system integrators of choice HSC works with global names to architect, integrate, deploy and manage their suite of OSS, BSS, VAS and IN in wireless (VoIP & IMS), wireline and hybrid networks.: NMS, Service Management & Provisioning .

DOMAIN EXPERTISE:

Terminals

- Terminal Platforms : iPhone, Android, Symbian, Windows CE/Mobile, BREW, PalmOS
- Middleware Experience & Applications : J2ME , IMS Client & OMA PoC,

Access

- Wired Access : PON & DSL, IP-DSLAM,
- Wireless Access : WLAN/WiMAX / LTE, UMTS, 2.5G, 2G ,Satellite Communication

Core Network

- IMS/3GPP , IPTV , SBC, Interworking , Switching solutions, VoIP

Applications

- Technologies : C, Java/J2ME, C++, Flash/lite,SIP, Presence, Location, AJAX/Mash
- Middleware: GlassFish, BEA, JBOSS, WebSphere, Tomcat, Apache etc.

Management & Back Office:

- Billing & OSS , Knowledge of COTS products , Mediation, CRM
- Network Management : NM Protocols, Java technologies,, Knowledge of COTS NM products, FCAPS, Security & Authentication

Platforms

- Embedded: Design, Development and Porting - RTOS, Device Drivers, Communications / Switching devices, Infrastructure components. Usage and Understanding of Debugging tools.
- FPGA & DSP : Design, System Prototyping. Re-engineering, System Verification, Testing

Automotive Telematics

- In Car unit (ECU) software design with CAN B & CAN C
- Telematics Network Design (CDMA, GSM, GPRS/UMTS)

BENEFITS

- **Reduced Time to market** : Complement your existing skills, Experience in development-to-deployment in complex communication systems, with key resources available at all times
- **Stretch your R&D dollars :** Best Shore" strategy to outsourcing**,** World class processes, Insulate from resource fluctuations

CONTACT INFORMATION:

Phone: +1.301.527.1629

Fax: +1.301.527.1690

eMail: whitepaper@hsc.com

Web: www.hsc.com

Blog : blog.hsc.com