1. INTRODUZIONE

Si vuole sviluppare un applicativo che implementi il protocollo di autenticazione OAuth2.

L'applicativo sarà composto dai seguenti componenti:

- Server di autenticazione (Core)
- Server delle risorse (Core)
- Client 1 (OAuth2 use case: authorization_code)
- Client 2 (OAuth2 use case: implicit)
- Client 3 (OAuth2 use case: password)
- Client 4 (OAuth2 use case: client_credentials)

Ognuno dei 4 client dovrà rappresentare uno use case del protocollo OAuth.

1.1 Glossario dei termini utilizzati

- **Resource owner**: l'utente proprietario delle risorse protette, che accede ad esse tramite un "Client" autorizzato.
- **Client**: un applicazione che che accede alle risorse protette per conto del "Resource owner".
- **Authorization server**: server che si occupa di rilasciare gli "Access Token" nel momento in vengono autenticati il "Client" e il "Resource owner" e la richiesta viene autorizzata.
- **Resource server (o Resource provider)**: il server che contiene le risorse protette e si occupa di accettare/fornire una risorsa utilizzando un "Access Token"
- **Access token**: token utilizzato per accedere alle risorse protette.

- **Authorization code**: un token intermedio, generato nel momento in cui il “Resource owner” autorizza un client ad accedere alle risorse protette. Il “Client” utilizzerà questo token per ottenere un “Access token”

1.2 Analisi dei requisiti

Capacità e possibilità tecniche.

Le varie web-app (**Client**) richiedono un basso livello di esperienza per essere utilizzata, gli utenti non hanno bisogno di essere guidati durante l’utilizzo.

Non è richiesta un ampia disponibilità di banda, infatti le risorse scambiate si limitano a semplici pagine web e brevi informazioni testuali (in formato JSON).

Le varie web-app (client) sono state progettate per essere utilizzate su qualunque device, tramite un web browser. Infatti sfruttando il Bootstrap di Twitter, sono state inserite delle regole per gestire display di diverse dimensioni.

Requisiti funzionali.

- L’**Authorization server** deve essere in grado di eseguire il processo di autenticazione (fornendo una pagina per la richiesta delle autorizzazioni e autenticando i vari client).
- L’**Authorization server** deve fornire agli utenti un sistema di gestione delle autorizzazioni concesse e delle applicazioni registrate.
- L’**Authorization server** deve essere in grado di stabilire se un’autorizzazione per accedere alle risorse protette è stata realmente concessa, se è scaduta o se è stata annullata.

- Il **Resource server** deve essere in grado di dialogare con l'Authorization server per verificare che l'access token (o l'autorizzazione) siano validi.
- Il **Resource server,** nel caso in cui l'autorizzazione sia valida, deve essere in grado di mettere a disposizione del Client le varie risorse protette.
- I vari **Client** devono permettere di portare a termine il processo di autenticazione, conservare l'Access token ottenuto dall'Authorization server e utilizzarlo per effettuare delle richieste al Resource server.

Requisiti non funzionali.

- Il **Resource owner** deve essere in grado di cambiare le proprie credenziali di accesso all'Authorization server.

1.3 Aspetti tecnologici

Lo standard utilizzato per lo scambio dei dati e delle risorse è il JSON.
E' stato preferito rispetto ad altre tecnologie per la sua semplicità nel rappresentare degli oggetti e scambiare informazioni.

Per la realizzazione dell'applicativo state utilizzate le seguenti tecnologie:

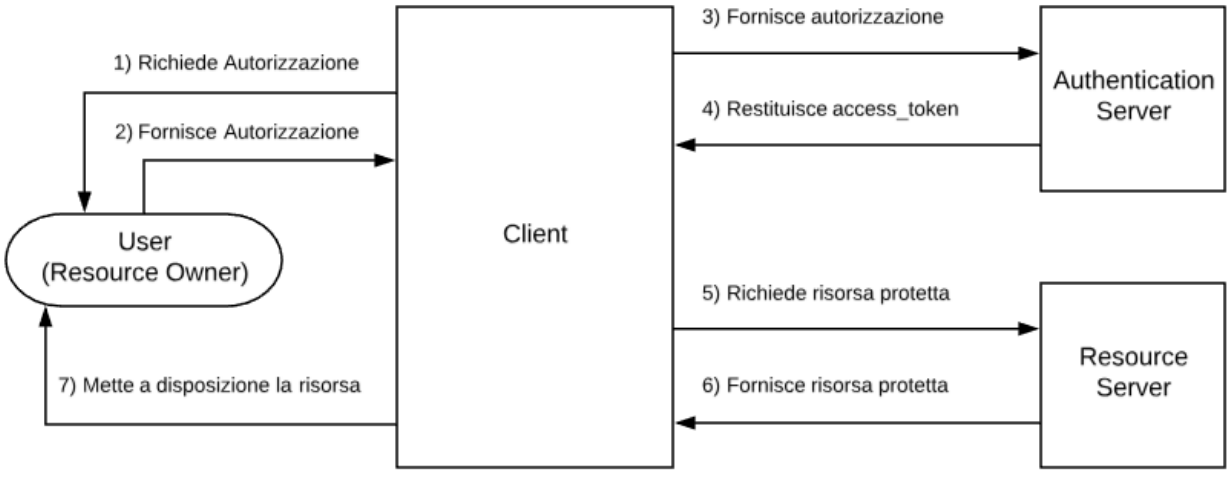
- **HTML5/CSS**:
  - Tutte le pagine sono state sviluppate in formato HTML5 utilizzando la libreria di templating "Handlebars" (NodeJS).
  - Per la realizzazione del layout è stata utilizzata la libreria del Bootstrap di Twitter in combinazione a delle regole di CSS personalizzare. Si è scelto di utilizzare il Bootstrap in quanto permette di creare rapidamente un design responsivo con elementi che sono già familiari agli utenti.

- **AJAX**:
  - Sono state implementate più chiamate di tipo GET e POST, per scambiare dati in formato JSON. E' stata preferita questa tecnologia per la sua facile implementazione e perché già presente nel Bootrap di Twitter.

- **NodeJS**:
  - E' stata utilizzata questa tecnologia per creare:
    - Il front-end e il back-end dell'Authentication Server.
    - Il front-end e il back-end del Resource Server.
    - Il front-end e il back-end dei vari client.

  - Principali moduli utilizzati:
    - Express: per un deploy rapido del back-end.
    - Handlebars: per il templating delle pagine.
    - Cookie-parser e express-session: per la gestione delle sessioni.
    - Body-parser e url: per ottenere le query o i parametri delle richieste.
    - Lowdb: utilizzato in sostituzione di un database relazionale.

1.4 Schema Funzionale (Generale)



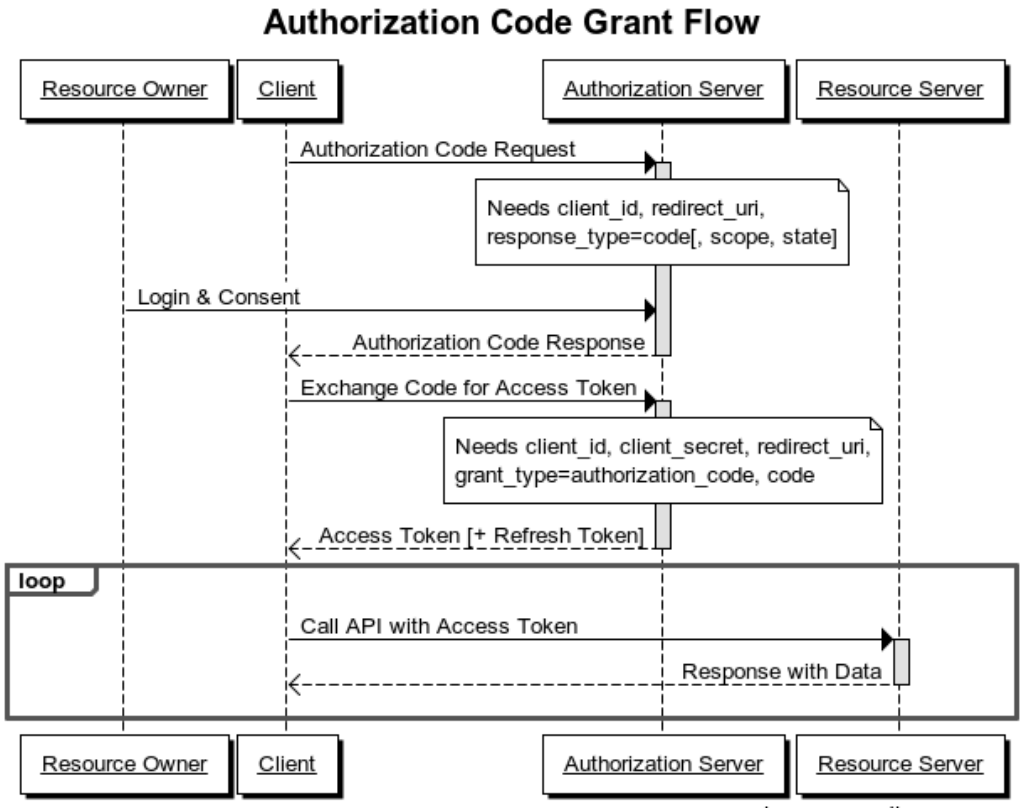
2. Flow di autenticazione

Sono stati realizzati 4 flow di autenticazione, ognuno per uno specifico use case del protocollo OAuth2. Di seguito viene analizzato ciascun flow.

2.1 Authorization Code (authorization code grant)

Il flow “Authorization Code” è lo use case più diffuso di OAuth2. Può essere utilizzato sia da client fidati che da client non fidati, per fornire un codice di autorizzazione (all’Authorization Server) detto **code** in cambio di un **access_token**.

In questo use case è previsto inoltre l’utilizzo di un **refresh_token**, al fine di rinnovare **access_token** dopo la sua scadenza.



1. Il client richiede il codice di autorizzazione (**code**) all’Authentication server tramite una GET all’end-point di autorizzazione. Vengono forniti:
    - **client_id**: l’identificativo con il quale è stata registrata l’applicazione (client).

- **redirect_url**: l’url di callback al quale verrà passato l’access_token (ad autorizzazione avvenuta)**.**
- **scope**: il livello di autorizzazione che l’utente decide di fornire all’applicazione (client).
- **state**: una stringa utilizzata dall’applicazione (client) come nonce (verrà verificata durante la richiesta del token).
- **response_type**: il tipo di risposta che l’applicazione (client) si aspetta (in questo caso **code**, ovvero il codice di autorizzazione).

2. L’Authentication server verifica se l’applicazione è registrata nel database e ri-direziona la richiesta su una pagina di autenticazione, dove l’utente può decidere se accettare o rifiutare, e il livello di autorizzazione che desidera fornire all’applicazione (client).

3. L’utente fornisce l’autorizzazione l’Authentication server genera un **code** (salvandolo nel database) e ri-direziona la richiesta sull’url di callback (**redirect_url**). Il client riceve così il codice il codice di autorizzazione (**code**) e il nonce (**state**).

4. Se il nonce **(state)** coincide a quello inviato nella richiesta di autorizzazione, il client esegue una richiesta POST all’end-point previsto per la richiesta del token sull’Authentication server fornendo i seguenti parametri:

- **client_id**: l’identificativo con il quale è stata registrata l’applicazione (client).
- **client_secret:** stringa di testo utilizzata come “password” dell’applicazione (Deve essere sempre mantenuta segreta)
- **redirect_url:** l’url di callback utilizzato in precedenza.
- **grant_type**: il tipo di autenticazione che il client sta chiedendo. (in questo caso: “authorization_code”)
- **code**: il codice di autorizzazione.

5. L’Authentication server verifica il **client_secret** nel database, controlla che il codice di autorizzazione **code** e l’url di callback utilizzato **redirect_url** siano identici a quelli utilizzati nella richiesta di autorizzazione.

   Se questa operazione ha esito positivo il server genera un **access_token** e un **refresh_token** e cancella il codice di autorizzazione dal database.

   Il server conclude la richiesta rispondendo con: l’**access_token** e il **refresh_token**.

   Il client riceve così l’**access_token** e il **refresh_token** un valore numerico che rappresenta la scadenza dell’**access_token.**

6. Il client utilizza l’**access_token** per accedere o modificare (in base al livello di autorizzazione) le risorse protette sul Resource Server.

7. Nel momento in cui l’**access_token** scade, il client può richiederne uno nuovo eseguendo una richiesta POST all’endpoint previsto per la richiesta del token sull’Authentication server fornendo i seguenti parametri:

   - **grant_type:** il tipo di autenticazione che il client sta chiedendo (in questo caso: “refresh_token”)
   - **client_id:** l’identificativo con il quale è stata registrata l’applicazione (client).
   - **client_secret:** stringa di testo utilizzata come “password” dell’applicazione (Deve essere sempre mantenuta segreta)
   - **refresh_token**: il token di “refresh” fornito precedentemente dal server.

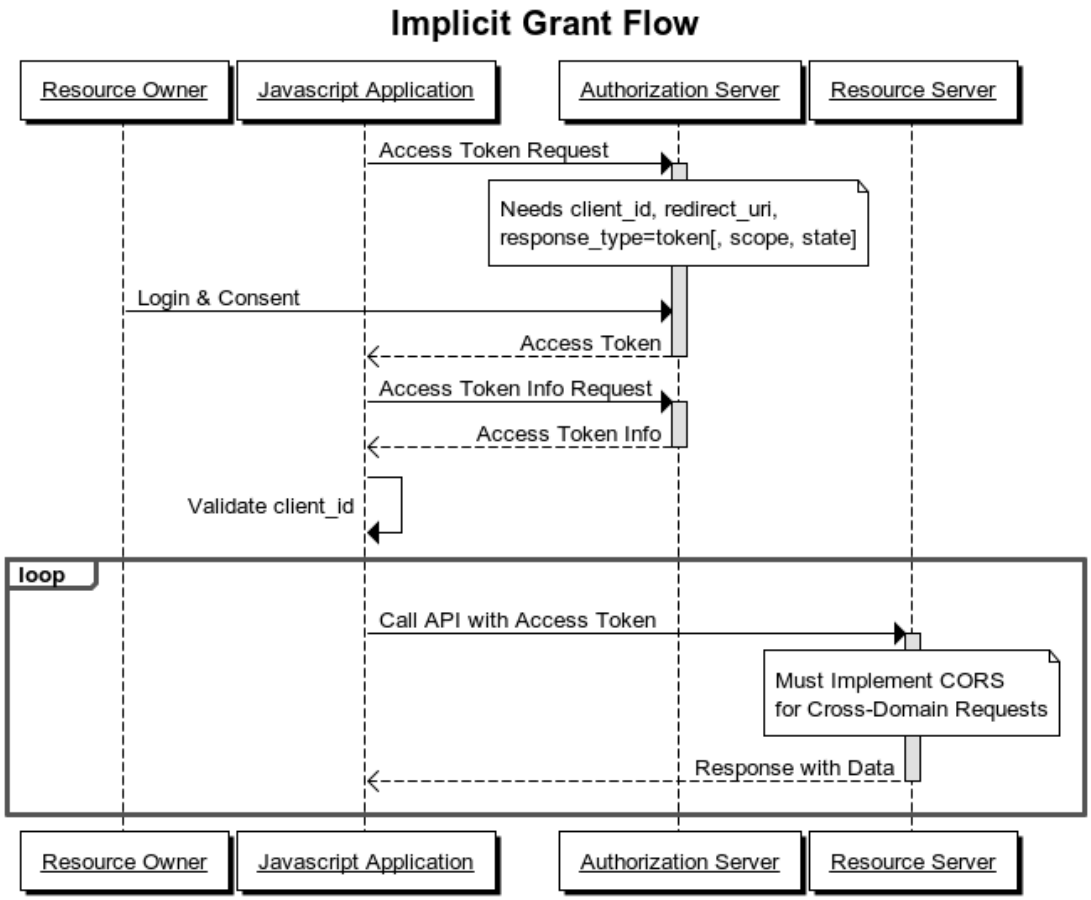
8. Se i parametri forniti sono corretti, l’Authentication server fornisce un nuovo **access_token**.

2.2 Implicit (implicit grant / single-page apps)

Il flow “Implicit” è una versione semplificata del flow “Authorization Code”, ottimizzata per client implementati in applicazioni per web-browsers (tipicamente scritti in javascript). Questo tipo di autorizzazione non prevede l’utilizzo di un **refresh_token** e non effettua nessuna richiesta all’end-point adibito alla distribuzione degli **access_token**.

L’**access_token** viene distribuito dall’Authentication passandolo come parametro al client sul proprio url di callback.

In questo tipo di autenticazione il client non viene autenticato dall’Authorization Server (infatti non viene scambiato il **client_secret**). In implementazioni dove è richiesto un alto livello di sicurezza può essere sfruttato il parametro **redirect_url** per compararlo con quello utilizzato per registrare l’applicazione.



1. Il client effettua una richiesta di tipo GET all’endpoint di autorizzazione dell’Autentication Server, passando come parametri:
    - **client_id**: l’identificativo con il quale è stata registrata l’applicazione (client).
    - **redirect_url**: l’url di callback al quale verrà passato l’access_token (ad autorizzazione avvenuta)**.**
    - **scope**: il livello di autorizzazione che l’utente decide di fornire all’applicazione (client).
    - **state**: una stringa utilizzata dall’applicazione (client) come nonce (verrà verificata durante la richiesta del token).
    - **response_type**: il tipo di risposta che l’applicazione (client) si aspetta (in questo caso **token**).

2. L’Authentication server verifica se l’applicazione è registrata nel database e ri-direziona la richiesta su una pagina di autenticazione, dove l’utente può decidere se accettare o rifiutare, e il livello di autorizzazione che desidera fornire all’applicazione (client).

3. Se l’utente accetta, l’Authentication server ri-direziona la richiesta sull’url di callback del client (**redirect_url)**. Il client riceve così il nonce (**state**), l’**access_token** e valore numerico che rappresenta la sua scadenza.

4. Se il nonce (**state**) coincide con quello della prima richiesta in client salva il token e utilizza l’**access_token** per accedere o modificare (in base al livello di autorizzazione) le risorse protette sul Resource Server.

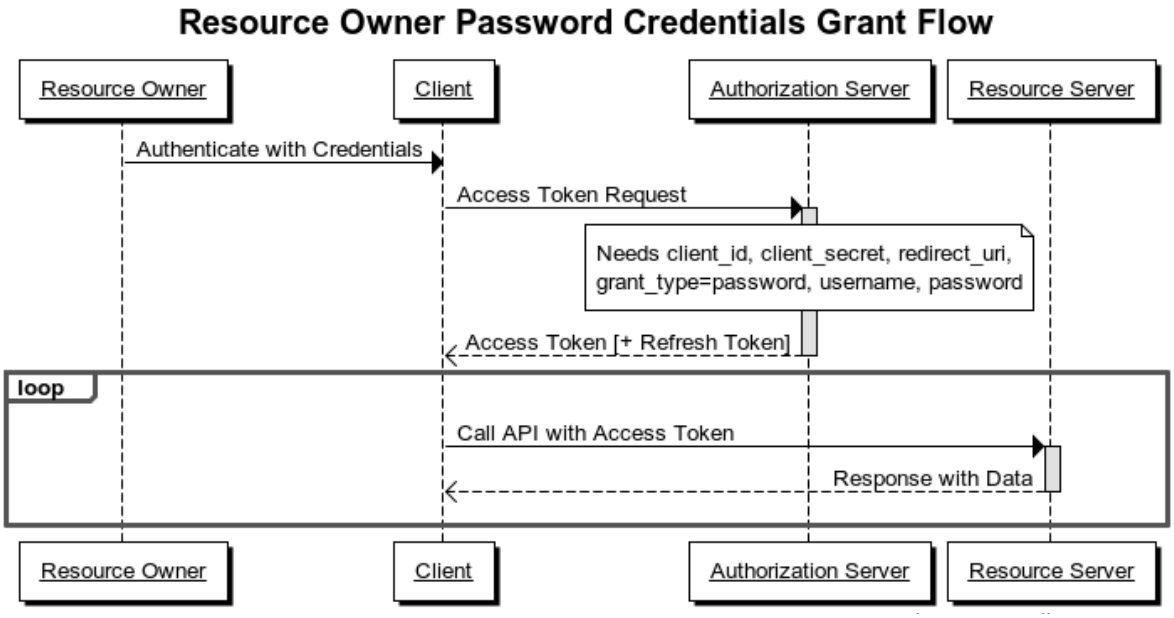
5. Allo scadere dell’**access_token** il client dovrà eseguire una nuova autenticazione. (infatti non è previsto il rinnovo dell’**access_token** in questo tipo di autorizzazione).

2.3 Password (resource owner password credentials grant)

Nel flow “Password” è possibile utilizzare le credenziali del Resource Owner (es. username e password) per permettere al client di ottenere un **access_token**.

Questo tipo di autorizzazione viene utilizzata solo in scenari dove vi è un alto livello di fiducia tra il Resource Owner e il Client (o se non è possibile implementare altri tipi di autenticazione).

Considerato che utilizzare le credenziali del Resource Owner è rischioso (dal punto di vista della sicurezza, se il client non è fidato), e che permettere al client di immagazzinare tali informazioni per riutilizzarle in futuro non è assolutamente accettabile, questo tipo di autorizzazione rilascia, oltre all’**access_token**, un **refresh_token** (per richiedere un nuovo **access_token** alla sua scadenza).



1. L’utente (Reource Owner) accede all’interfaccia di log-in e si autentica con le sue credenziali direttamente sul client.

2. Il client esegue una richiesta POST all'end-point previsto per la richiesta del token sull'Authentication server fornendo i seguenti parametri:
   - **client_id**: l'identificativo con il quale è stata registrata l'applicazione (client).
   - **client_secret:** stringa di testo utilizzata come "password" dell'applicazione (Deve essere sempre mantenuta segreta)
   - **redirect_url:** l'url di callback utilizzato in precedenza.
   - **username**: lo username che il Resource Owner utilizza per autenticarsi con l'Authorization Server.
   - **password**: la password che il Resource Owner utilizza per autenticarsi con l'Authorization Server.
   - **grant_type**: il tipo di autenticazione che il client sta chiedendo. (in questo caso: "**password**")
   - **scope**: il livello di autorizzazione fornito all'applicazione (client). In questo tipo di autorizzazione è implicito e non può essere modificato dall'utente finale.

   Il client deve essere configurato in modo da non loggare/salvare mai i parametri **username** e **password**.

3. L'Authentication server verifica se l'applicazione è registrata nel database e se le credenziali dell'applicazione (**client_id** e **client_secret**) corrispondono.

   Se questa operazione ha esito positivo il server genera un **access_token** e un **refresh_token**.

   Il server conclude la richiesta rispondendo con: l'**access_token** e il **refresh_token**.

   Il client riceve così l'**access_token** e il **refresh_token** un valore numerico che rappresenta la scadenza dell'**access_token**.

4. Il client utilizza l’**access_token** per accedere o modificare (in base al livello di autorizzazione) le risorse protette sul Resource Server.

5. Nel momento in cui l’**access_token** scade, il client può richiederne uno nuovo eseguendo una richiesta POST all’endpoint previsto per la richiesta del token sull’Authentication server fornendo i seguenti parametri:

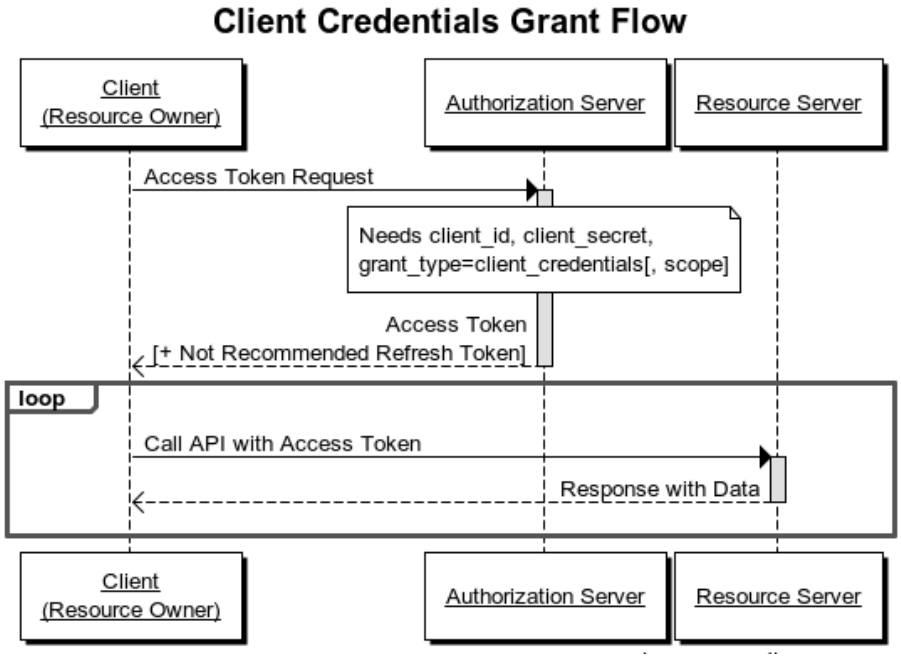
   - **grant_type:** il tipo di autenticazione che il client sta chiedendo (in questo caso: “refresh_token”)
   - **client_id:** l’identificativo con il quale è stata registrata l’applicazione (client).
   - **client_secret:** stringa di testo utilizzata come “password” dell’applicazione (Deve essere sempre mantenuta segreta)
   - **refresh_token**: il token di “refresh” fornito precedentemente dal server.

   Se i parametri forniti sono corretti, l’Authentication server fornisce un nuovo **access_token**.

2.4 Client Credentials (client credentials grant)

Il flow “Client Credentials” è l’implementazione più semplificata di OAuth2. Il client viene autorizzato utilizzando unicamente il suo **client_id** e **client_secret**. Questo tipo di autorizzazione può essere utilizzata per eseguire autenticazioni di tipo “machine-to-machine”, dove un’autorizzazione da parte dell’utente (Resource Owner) non è necessaria.

In questo tipo di autorizzazione non raccomandato l’uso di un **refresh_token**, quindi alla scadenza dell’**access_token** il client dovà ri-eseguire la procedura di autenticazione.



1. Il client esegue una richiesta POST all’end-point previsto per la richiesta del token sull’Authentication server fornendo i seguenti parametri:
    - **client_id**: l’identificativo con il quale è stata registrata l’applicazione (client).
    - **client_secret:** stringa di testo utilizzata come “password” dell’applicazione (Deve essere sempre mantenuta segreta)
    - **grant_type**: il tipo di autenticazione che il client sta chiedendo. (in questo caso: “**client_credentials**”)

- **scope**: il livello di autorizzazione fornito all’applicazione (client). In questo tipo di autorizzazione è implicito e non può essere modificato dall’utente finale.

2. L’Authentication server verifica se l’applicazione è registrata nel database e se le credenziali dell’applicazione (**client_id** e **client_secret**) corrispondono.

Se questa operazione ha esito positivo il server genera un **access_token**.

Il server conclude la richiesta rispondendo con: l’**access_token**

Il client riceve così l’**access_token** e un valore numerico che rappresenta la sua scadenza.

3. client salva il token e utilizza l’**access_token** per accedere o modificare (in base al livello di autorizzazione) le risorse protette sul Resource Server.

4. Allo scadere dell’**access_token** il client dovrà eseguire una nuova autenticazione. (infatti non è previsto il rinnovo dell’**access_token** in questo tipo di autorizzazione).

3. ACCESS-TOKEN

E’ stato realizzato un apposito modulo per la generazione, verifica e validazione dell’access-token (oauth2_lib.js), che viene incluso nell’applicativo dell’Autentication Server.

3.1 Inizializzazione

```
config.oauth_hmac_crypt = 'MpetZfi6yNJWnwWxC4r2HcpfpHoZLVGB'
config.oauth_hmac_sign =    't7R6NsjynLWb8GZLVgGxGxaHnpik7Kxg';
```

File in immagine: oauth_authorization_server/config.js

```
var oauth2_lib = require('./oauth2_lib').oauth2_lib
var oauth2 = new oauth2_lib({crypt_key: config.oauth_hmac_crypt, sign_key: config.oauth_hmac_sign})
```

File in immagine: (generico file dove viene utilizzata la libreria)

```
// Classe + Costruttore
function oauth2_lib(crypt_key, sign_key) {

    // Inizializzazione di serializer (HMAC AES256)
        // crypt_key    -> chiave di cifratura
        // sign_key     -> firma digitale
    this.serializer = serializer.createSecureSerializer(crypt_key, sign_key);

    // c = this.serializer.stringify("prova")   -> cifratura
    // m = this.serializer.parse(s)             -> decifratura

    this.access_token_expiration = config.access_token_expiration;
    this.refresh_token_expiration = config.refresh_token_expiration;
}

// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

oauth2_lib.prototype.serialize_string = function(string) {
        var c = this.serializer.stringify(string)
        return c
};

oauth2_lib.prototype.parse_string = function(user_id, client_id, scope) {
        var m = this.serializer.parse(string)
        return m
};
```

File in immagine: oauth_authorization_server/oauth2/oauth2_lib.js

La libreria viene inclusa e vengono passati al costruttore una chiave di cifratura e una chiave per verificare la firma della cifratura (viene utilizza infatti una funziona HMAC per cifrare il valore del token)

3.2 Generazione dell'access_token

```
oauth2_lib.prototype.generateAccessToken = function(prms) {

        var token = {};

        //if (prms['grant_type'] != undefined)  token ['grant_type'] = prms['grant_type'];
        if (prms['code'] != undefined)  token ['code'] = prms['code'];
        //if (prms['redirect_uri'] != undefined)  token ['redirect_uri'] = prms['redirect_uri'];
        if (prms['user_id'] != undefined)  token ['user_id'] = prms['user_id'];
        if (prms['client_id'] != undefined)  token ['client_id'] = prms['client_id'];
        if (prms['client_secret'] != undefined)  token ['client_secret'] = prms['client_secret'];
        if (prms['scope'] != undefined)  token ['scope'] = prms['scope'];

        var now = (Math.floor(Date.now() / 1000));
        var expire = (now + this.access_token_expiration);

        token['expire'] = expire;

        var c = this.serializer.stringify(token);

        return c;

}; // Ritorna una stringa che viene utilizzata come "access token"
```

La funzione di generazione accetta come parametro un dizionario di tipo **chiave: valore.** In base al tipo di implementazione scelta si potranno passare nel dizionario le seguenti informazioni:

- *grant_type*
- code
- *redirect_uri*
- user_id
- client_id
- client_secret
- scope

(Nella realizzazione del progetto sono stati rimossi rimossi i parametri in *corsivo* per non impattare sulla lunghezza del token)

La funzione non implica che tutti i parametri siano passati obbligatoriamente. Durante la realizzazione del token viene aggiunto il parametro **expire** che definisce la scadenza del token.

Il dizionario risultante viene cifrato e serializzato, ottenendo così una stringa di caratteri.

3.3 Validazione dell'access_token

```
oauth2_lib.prototype.validateAccessToken = function(access_token, scope) {
        var m = this.serializer.parse(access_token);

        var now = (Math.floor(Date.now() / 1000));
        var expired = true;                                          // Verifica se l'access token è scaduto
        if ( now < m['expire'] ) { expired = false } else { expired = true }


        if ( m['scope'].includes(scope) && expired == false) {
            return true;
        } else {
            return false;
        }
};  // Ritorna vero se l'access token e lo scope sono validi
```

Per validare l'access_token è sufficiente provare a decifrarlo e deserializzarlo. Se il token è legittimo il risultato sarà un dizionario di parametri.

Vengono inoltre controllati e confrontati i seguenti parametri:

- **expire** permette di verificare che il token non sia scaduto
- **scope** permette di verificare che il token sia valido per il livello di autorizzazione che l'utente sta richiedendo.

La funzione restituisce **true** se l'**access_token** è valido, in caso contrario restituisce **false**.

3.4 Generazione del refresh_token

La funzione di generazione del **refresh_token** prendendo in input l'**access_token** genera una stringa composta da:

```
oauth2_lib.prototype.generateRefreshToken = function(access_token) {
        var m = this.serializer.parse(access_token);

        var refresh_token = {};

        var now = (Math.floor(Date.now() / 1000));
        var expire = (now + this.refresh_token_expiration);

        refresh_token ['client_id'] = m['client_id'];
        refresh_token ['expire'] = expire;

        var c = this.serializer.stringify( refresh_token );
        return c
};
```

- **client_id**: l'identificativo con il quale è stata registrata l'applicazione (client).
- **expire**: definisce la scadenza del **refresh_token**.

3.5 Ri-generazione dell’access_token

```
oauth2_lib.prototype.refreshAccessToken = function(access_token, ref_access_token) {
        var m1 = this.serializer.parse(access_token);
        var m2 = this.serializer.parse(ref_access_token);

        var now = (Math.floor(Date.now() / 1000));
        var expired = true;                                          // Verifica se il refresh token è scaduto
        if ( now < m2['expire'] ) { expired = false } else { expired = true }


        if ( m1['client_id'] == m2['client_id'] && expired == false) {      // Se i due client id corrispondono e il refresh token non è scaduto
            var now = (Math.floor(Date.now() / 1000));
            var new_expire = now + this.access_token_expiration;

            m1 ['expire'] = new_expire;
            var c = this.serializer.stringify( m1 );
            console.log(c)
            return c

        } else {
            return undefined;
        }

};
```

La funzione di “refresh” dell’**access_token** prende in input l’**access_token** scaduto e il **refresh_token** corrispondente.

La funzione verifica che il **refresh_token** non sia scaduto e che i due **client_id** sono identici.

Se l’operazione da esito positivo viene aggiornato il parametro **expire** e viene generato il nuovo **access_token**.

4. Codice

Vengono riportati frammenti del codice più significativo:

```
authorize_fx: function  (req, res, next){
    // Parso la query
    var url_parts = url.parse(req.url, true);
    var query = url_parts.query;

    // Prendo i parametri della query
    var response_type = (req.query.response_type || req.body.response_type);        // Il tipo di riposta che il client si aspetta
    var client_id = (req.query.client_id || req.body.client_id);                    // L'id del client
    //var client_secret = (req.query.client_secret || req.body.client_secret);      // Il secret del client
    //var redirect_uri = (req.query.redirect_uri || req.body.redirect_uri);         // L'url di callback del client (non usato nel flow client_crede
    //var scope = (req.query.scope || req.body.scope);                              // scope
    //var state = (req.query.state || req.body.state);                              // Stringa utilizzata come nonce dal client

    // Controllo che i parametri comuni siano stati forniti
    if (!response_type || !client_id){
        res.status(400);
        res.end( JSON.stringify({ "status": "error", "message": "sono richiesti response_type, client_id" }) );
        return;
    }

    // Controllo che sia registrata l'applicazione con id client_id
    var app = find_client(client_id)
    if (!app) {
        res.status(400);
        res.end( JSON.stringify({ "status": "error", "message": "l\'applicazione con id: " + client_id + " non è registrata" }) );
        return;
    }


    // Controllo che l'app sia registrata per il tipo di autenticazione richiesto
    if (app['grant_type'] != from__response_type__to__grant_type(response_type)){
        res.status(400);
        res.end( JSON.stringify({ "status": "error", "message": "l\'applicazione con id: " + client_id + " prevede un tipo di autenticazione diverso
        return;
    }

    console.log("\033[0;35m[*] Client: " + client_id + " requested authorization (" + response_type + ")\033[0m")

    // Scelgo flow
    switch(response_type) {
        case "code":                    aut_flow_1(req, res, next, app); break;             // flow authorization_code_grant
        case "token":                   aut_flow_2(req, res, next, app); break;             // flow implicit_grant
        //case "password":              aut_flow_3(req, res, next, app); break;             // flow resource_owner_password_credentials_grant
        //case "client_credentials":    aut_flow_4(req, res, next, app); break;             // flow client_credentials_grant
    }
}
```

Selettore delle autorizzazioni.

```
if (!client_id || !redirect_uri || !scope || !state){
    res.status(400);
    res.end( JSON.stringify({ "status": "error", "message": "sono richiesti client_id, redirect_uri, scope, state" }) );
    return;
}

if (! scope.includes("basic") ){
    res.status(400);
    res.end( JSON.stringify({ "status": "error", "message": "lo scope deve essere almeno di tipo basic" }) );
    return;
}


// Creo il codice e lo metto nel database
var code = randomstring.generate(8)
var code_expire = (Math.floor(Date.now() / 1000) + 60*5); // Il codice vale per 5 minuti.

db.read()   // Refresho la cache
db.get('codes')
.push({
    "code": code,
    "client_id": client_id,
    "expire": code_expire,
    "redirect_uri": redirect_uri
})
.write()


var request_permission = get_request_permission (scope, true);
var callback_url = redirect_uri + "?code=" + code + "&state=" + state + "&scope="

res.render("authorize", {
        "customizable": true,
        "callback_url": callback_url,
        "request_permission": request_permission,
        "client_name": app['client_name']
    })
```

Frammento del flow di autorizzazione “**authorization_code**”

```
token_fx: function  (req, res, next){
        // Parso la query
    var url_parts = url.parse(req.url, true);
    var query = url_parts.query;

    // Prendo i parametri della query
    var grant_type = (req.query.grant_type || req.body.grant_type);                 // Il tipo di riposta che il client si aspetta
    var client_id = (req.query.client_id || req.body.client_id);                    // L'id del client
    //var client_secret = (req.query.client_secret || req.body.client_secret);      // Il secret del client
    //var redirect_uri = (req.query.redirect_uri || req.body.redirect_uri);         // L'url di callback del client (non usato nel flow client_crede
    //var code = (req.query.code || req.body.code);                                 // Stringa utilizzata come nonce dal client
    //var scope = (req.query.scope || req.body.scope);                              // scope

    // Controllo che i parametri comuni siano stati forniti
    if (!grant_type || !client_id ){
        res.status(400);
        res.end( JSON.stringify({ "status": "error", "message": "sono richiesti grant_type, client_id" }) );
        return;
    }

    // Controllo che sia registrata l'applicazione con id client_id
    var app = find_client(client_id)
    if (!app) {
        res.status(400);
        res.end( JSON.stringify({ "status": "error", "message": "l\'applicazione con id: " + client_id + " non è registrata" }) );
        return;
    }


    // Controllo che l'app sia registrata per il tipo di autenticazione richiesto
    /*if (app['grant_type'] != grant_type){
        res.status(400);
        res.end( JSON.stringify({ "status": "error", "message": "l\'applicazione con id: " + client_id + " prevede un tipo di autenticazione diverso
        return;
    }*/     // <- Commentata perchè non mi permette di gestire il caso refresh_token

    console.log("\033[0;35m[*] Client: " + client_id + " requested token (" + grant_type + ")\033[0m")

    // Scelgo flow
    switch(grant_type) {
        case "refresh_token":           tk_refresh(req, res, next, app); break;             // flow refresh_token
        case "authorization_code":      tk_flow_1(req, res, next, app); break;              // flow authorization_code_grant
        //case "implicit"               tk_flow_2(req, res, next, app); break;              // flow implicit_grant
        case "password":                tk_flow_3(req, res, next, app); break;              // flow resource_owner_password_credentials_grant
        case "client_credentials":      tk_flow_4(req, res, next, app); break;              // flow client_credentials_grant
    }
}
```

Selettore dei token.

```
// Cerco il codice nel database - - - - - - - - - - -
db.read()   // Refresho la cache
var fcode = db.get('codes')
    .find({ "client_id": client_id, "code": code, "redirect_uri":redirect_uri })
    .value()

if(!fcode){
    res.status(400);
    res.end( JSON.stringify({ "status": "error", "message": "parametro code non valido" }) );
    return;
}


// Genero e salvo l'access token - - - - - - - - - -
var access_token = oauth2.generateAccessToken({ "code":code, "client_id": client_id, "scope": scope  });
var refresh_token = oauth2.generateRefreshToken(access_token);

if(!access_token || !refresh_token){
    res.status(400);
    res.end( JSON.stringify({ "status": "error", "message": "errore nella generazione dell\'access_token" }) );
    return;
}

db.read()   // Refresho la cache
    db.get('authorizations')
    .push({
        "auth_id": randomstring.generate(8),
        "issued_at": (Math.floor(Date.now() / 1000)),
        "expire_at": (Math.floor(Date.now() / 1000)) + config.access_token_expiration,
        "expire_in": config.access_token_expiration,
        "owner": app['owner'],
        "grant_type": grant_type,
        "client_id": client_id,
        "scope": scope,
        "access_token": access_token,
        "refresh_token": refresh_token
    })
    .write()


// Rimuovo il code - - - - - - - - - - - - - - - - -
db.read()   // Refresho la cache
db.get('codes')
    .remove({ "client_id": client_id, "code": code })
    .write()

res.end( JSON.stringify({ "status": "ok", "token_type":"bearer", "access_token":access_token, "expires":config.access_token_expiration, "refresh_tok
return;
```

Frammento della generazione del token nel flow “**authorization_code**”

5. Conclusioni

Grazie a questo progetto è stato possibile comprendere il funzionamento di OAuth2, apprezzandone i vantaggi relativi all'autenticazione e allo scambio di informazioni fra client, authentication server e resource provider.

L'utilizzo di OAuth2 ci permette infatti di delegare il processo di autenticazione a un server trusted (Authentication server). Grazie a questa architettura è inoltre possibile definire uno schema per permettere all'utente finale (Resource owner) di stabilire il livello di autorizzazione che un Client ha sulle proprie informazioni conservare su un Resource Server.

6. Nota bibliografica e sitografica

Vengono elencate di seguito le principali documentazioni utilizzate per la realizzazione del progetto:

1. Pagina ufficiale di OAuth:
   https://oauth.net/2/

2. Documentazione ufficiale di OAuth:
   https://tools.ietf.org/html/rfc6749

3. Guide di riferimento:
   - http://www.bubblecode.net/en/2016/01/22/understanding-oauth2
   - https://aaronparecki.com/oauth-2-simplified/
   - https://connect2id.com/products/server/docs/api/token
   - https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2