Guida all'estrazione di dati dai *Social Network*

Di Donato Leonardo

19 luglio 2013

Indice

Elenco delle figure

Elenco delle tabelle

Sommario

Questo articolo affronta l'estrazione dei dati dai due maggiori *Social Network* attualmente presenti: Facebook e Twitter.

Con l'avvento dei *Social Network* ogni giorno viene generata e memorizzata una quantità enorme di informazioni. I *Social Network* rappresentano quindi una fonte potenzialmente infinita di dati degli utenti, i quali possono essere sfruttati sia a scopi scientifici sia a scopi commerciali.

I *Social Network*, al fine di permettere lo sviluppo di un ecosistema di applicazioni attorno alla propria piattaforma, mettono a disposizione delle interfacce per permettere l'accesso ai dati dei propri utenti (nel rispetto della loro privacy). Tuttavia, poiché essi fondano il loro stesso modello di business su tali dati, ne consegue che le modalità di accesso ai dati e la quantità di dati estraibili dai *Social Network* è molto ridotta.

In questo articolo si affrontano dapprima le modalità di accesso legale ai dati di Facebook e Twitter, presentando degli esempi pratici, e successivamente si affronta una discussione sulle modalità alternative (solitamente associate al *Web Scraping*) di estrazione dei dati dai *Social Network*.

Si evidenziano perciò le limitazioni quantitative e qualitative delle interfacce d'accesso ai dati dei *Social Network* così come le limitazioni tecniche che permettono o impediscono l'estrazione dei dati non autorizzata.

1 Introduzione

L'estrazione dei dati dai *Social Network* è attualmente uno degli argomenti di maggior interesse. Infatti, tali informazioni offrono molte opportunità di ricerca e analisi: analizzando i dati di una rete sociale è possibile investigare le risposte a molti quesiti inerenti le reti sociali (i. e., una rappresentazione approssimata del mondo reale).

Ad esempio, di seguito si presentano alcuni quesiti che è possibile investigare tramite l'estrazione di informazioni dai *Social Network* e la loro successiva analisi.

- Quanto frequentemente un determinato utente comunica con un altro?
- Quanto è simmetrica la comunicazione fra gli utenti di una rete sociale?
- Quali sono gli utenti più influenti (e/o popolari)?
- Quali sono gli argomenti principalmente affrontati e discussi da un utente?
- Quali sono gli argomenti verso i quali un determinato insieme (e. g., geografico) di utenti prova maggiore interesse?
- Qual è l'opinione di un utente o un insieme di utenti relativamente a un argomento?

I *Social Network* presi in esame per questo articolo sono Facebook e Twitter.

Facebook è una rete sociale che permette agli utenti registrati di creare un proprio profilo in cui caricare foto, video e con il quale inviare messaggi, al fine di mantenersi in contatto con amici, familiari e colleghi. Possiede inoltre le seguenti funzionalità: gruppi, eventi, pagine, *chat*, *marketplace* e gestore della *privacy*.

Con oltre 1 miliardo di utenti (al 31/03/2013[1]), 300 milioni di immagini caricate giornalmente, quasi 3 miliardi di *like* giornalieri e 2,5 miliardi di comunicazioni giornaliere (al 15/06/2013[2]), è il principale *Social Network*.

Twitter è invece un servizio di *micro-blogging* con due caratteristiche principali:

- i suoi utenti inviano messaggi (i. e., *tweet*) di massimo 140 caratteri composti solitamente da parole chiave (sotto forma di *hashtag*), linguaggio naturale e abbreviazioni comuni
- ogni utente può seguire (*follow*) altri utenti affinché la propria *timeline* sia popolata dai loro *tweet*.

Twitter conta oltre 500 milioni di utenti, quasi 60 milioni di *tweet* giornalieri e oltre un miliardo di comunicazioni giornaliere (al 05/07/2013[3]).

[1] http://investor.fb.com/releasedetail.cfm?ReleaseID=761090
[2] http://en.kioskea.net/faq/26769-facebook-some-statistics-on-the-daily-data-transfers
[3] http://www.statisticbrain.com/twitter-statistics

Nel momento in cui si stila questo documento, entrambi i *Social Network* in questione mettono a disposizione delle `REST API` con meccanismo di autenticazione *OAuth*.

In questo articolo si presentano le funzionalità di tali `API` e, in alcuni casi, se ne mostra il funzionamento tramite esempi pratici realizzati nel linguaggio `R` (R Core Team 2013) con l'ausilio di librerie esterne, quali ad esempio `rjson` (Couture-Beil 2013) e `RCurl` (Lang 2013).

Prima di addentrarsi nell'esplorazione del meccanismo di accesso ai dati per entrambi i *Social Network* è indispensabile presentare i concetti basilari dei *framework* su cui tali meccanismi sono costruiti.

1.1 API

Con il termine *Application Programming Interface* (`API`) si indica un insieme di procedure rese disponibili all'esterno, di solito raggruppate a formare un insieme di strumenti specifici per l'espletamento di un determinato compito all'interno di un certo programma.

Tale termine si riferisce a un concetto molto generale. La finalità delle `API`, generalmente, consiste nel fornire un livello di astrazione tra un servizio (livello inferiore) e il suo fruitore (che può essere a sua volta un altro servizio, un altro software, etc. etc.). Il loro concetto rientra, quindi, nel più vasto concetto di riuso del software.

Come parzialmente intuibile dai termini che compongono l'acronimo, le `API` rappresentano un'**interfaccia** di programmazione. Tale interfaccia ha lo scopo di permettere ad altre entità (e. g., librerie, software, utenti) di compiere un insieme di azioni su una determinata piattaforma di cui non si conoscono i dettagli implementativi. Per tale motivo, le `API` vengono spesso fornite per permettere oltre che l'utilizzo di un determinato servizio, anche la sua estensione da parte di altri attori. Mettere a disposizione delle `API` di un software significa dare ad altri la possibilità di interagire con la piattaforma di tale software e, eventualmente, di estendere le funzioni e le caratteristiche della sua struttura base. In altri termini, le `API` sono lo strumento primario utilizzato per permettere l'interazione ad alto livello con i software (o, generalmente, con un'implementazione a più basso livello).

Tutti i maggiori *Social Network* esistenti forniscono, infatti, delle `API`.

Quando usate nel contesto *Web*, le `API` sono tipicamente definite come un insieme di possibili richieste `HTTP` che restituiscono un messaggio di risposta con una struttura ben definita (`XML` o `JSON`, solitamente). Anche se, storicamente, le `API` sul *Web* sono nate e pensate come servizi *Web*, ad esempio di tipo `SOAP` (*Simple Object Access Protocol*) o di tipo `SOA` (*Service-oriented Architecture*), attualmente questo paradigma è stato ripensato in favore di un approccio più diretto nella rappresentazione dello stato di trasferimento, a cui ci si riferisce con il termine *restful* `API` (`REST`).

1.1.1 REST API

Con il termine `REST` (*Representational State Transfer*) si indica un'architettura, finalizzata alla creazione di applicazioni di rete, basata su un protocollo di comunicazione

client-server senza stato. Nella quasi totalità dei casi tale protocollo corrisponde al protocollo su cui si basa l'architettura *Web*, cioè il protocollo `HTTP`. Tuttavia è importante specificare che tale architettura è indipendente dal protocollo poiché con essa si interfaccia, non identifica (Fielding 2000).

Quindi, solitamente, l'idea fondamentale di tale approccio consiste nell'utilizzare un protocollo di comunicazione, ad esempio il protocollo `HTTP`, per far comunicare due macchine su una rete. Questo approccio si identifica quindi come un'alternativa, ormai preponderante, a meccanismi quali le `RPC` (*Remote Procedure Calls*) e i servizi *Web* (e. g., `WSDL`, `SOAP`[4]). Le applicazioni basate su tale approccio utilizzano quindi il protocollo `HTTP` per inviare (creandoli o aggiornandoli qualora già esistenti), leggere e cancellare i dati. Ne consegue che le `API REST` utilizzando il protocollo `HTTP` per tutte le operazioni CRUD[5]. Si osservi che uno dei principi primari dell'architettura `REST` consiste nel fatto che ogni risorsa deve essere identificata da un `URI` univoco.

Oltre al livello di astrazione che permette di raggiungere, questo stile architetturale ha altri punti di forza; infatti, anche essendo molto semplice da rispettare e implementare, esso non è sprovvisto di alcuna funzionalità e possibilità.

Fielding (2000) ha posto i seguenti principi alla base dell'approccio `REST`:

1. interfaccia uniforme (e. g., ogni risorsa identificata univocamente, gli `URI` delle risorse devono essere contenuti nel corpo o nell'intestazione delle risposte);
2. mancanza di stato, nel senso che lo stato è contenuto nelle richiesta stessa (e. g., sotto forma di parametro, contenuto del corpo della richiesta);
3. caching;
4. separazione tra *client* e *server*;
5. sistema livellato (un *client* non può e non deve sapere a quale *server* è connesso).

Si mostra infine un breve esempio finalizzato a chiarire la differenza tra una `API SOAP` e una `API` implementata tramite lo stile architetturale `REST`. Mentre nel primo caso il servizio *Web* `SOAP` modella lo scambio tra *client* e *server* come chiamate ad oggetti, nell'approccio `REST` ciò viene astratto (e delegato allo standard di comunicazione) poiché nessun metodo viene esposto.

Di seguito un esempio di una ipotetica chiamata ad una `API SOAP`:

```
exampleService.getResource(1)
```

Qualora la stessa `API` venga implementata con approccio `REST`, la chiamata precedente verrebbe tradotta in una chiamata `HTTP GET` al seguente `URI`:

```
http://example.com/product/1
```

[4] Meccanismo per lo scambio remoto di messaggi.
[5] *Create/Read/Update/Delete.*

1.2 JSON

`JSON` è un formato utilizzato per lo scambio di dati. Costituisce, inoltre, un sottoinsieme della *JavaScript's Object Notation*, cioè il modo in cui gli oggetti vengono costruiti in `JavaScript` (Crockford 2006).

`JSON` è costituito da due sole strutture:

- una collezione di coppie (nome, valore) che ha il vantaggio di poter essere tradotta facilmente in molti linguaggi in molti modi (i. e., un oggetto, un *record*, uno *struct*, un dizionario, una lista con chiavi o un *array* associativo)
- una lista ordinata di valori, rappresentabile in molti linguaggi come una lista, un *array*, o una sequenza.

Come già anticipato, `JSON` è il formato maggiormente utilizzato per i messaggi di risposta dei servizi *Web*, quindi anche delle `API`. Ha soppiantato il formato `XML` in tale ruolo poiché è molto più leggero, anche se pur caratterizzato dallo stesso potere espressivo.

1.3 OAuth

Il termine *OAuth* si riferisce a un protocollo generico di autenticazione aperta. Lo scopo che tale protocollo si prefigge è quello di fornire un *framework* per la verifica delle identità delle entità coinvolte in transazioni sicure.

Esistono, al momento, due versioni di questo protocollo: *OAuth 1.0a*[6] (Hammer-Lahav 2010) e *OAuth 2.0* (Hardt 2012). Entrambe le versioni supportano l'autenticazione *two-legged* (letteralmente, "a due gambe"), in cui un *server* viene garantito circa l'identità dell'utente, e l'autenticazione *three-legged*, in cui un *server* è garantito da un'applicazione (o, più generalmente, da un *content provider*) circa l'identità dell'utente. Quest'ultimo tipo di autenticazione richiede l'utilizzo degli *access token* ed è quella comunemente implementata dai *Social Network* (e. g., Facebook e Twitter) attualmente.

1.3.1 Autenticazione three-legged

Il punto focale della specifica *OAuth* è che il *content provider* (e. g., un'applicazione Facebook) deve garantire al *server* che il *client* possegga un'identità. L'autenticazione *three-legged* offre tale funzionalità senza che il *client* o il *server* necessitino mai di conoscere i dettagli di tale identità (i. e., *username* e *password*).

Il processo di autenticazione *three-legged* funziona tramite i seguenti passi:

1. Il *client* effettua una richiesta di autenticazione al *server*, il quale controlla che il *client* sia un utente legittimo del servizio che esso offre

[6] Il suffisso "a" sta ad indicare una miglioria per incrementare il livello di sicurezza apportata alla prima versione di *OAuth*.

2. Il *server* indirizza il *client* verso il *content provider* affinché possa richiedere l'accesso alle sue risorse

3. Il *content provider* valida l'identità del *client* e (spesso) richiede i permessi necessari ad accedere ai suoi dati

4. Il *content provider* indirizza il *client* verso il *server*, notificando il successo o il fallimento della sua operazione. Questa operazione, che è anch'essa una richiesta, include un codice di autorizzazione nel caso di successo dell'operazione precedente.

5. Il *server* effettua una richiesta *out-of-band*[7] al *content provider*, scambiando il codice di autorizzazione ricevuto con un *access token*.

Si noti come il *server* verifichi sia l'identità dell'utente (i. e., *client*), sia quella del consumatore (i. e., *content provider*).

Ogni scambio (i. e. *client* verso *server* e *server* verso *content provider*) include la validazione di una chiave segreta condivisa.

La differenza tra *OAuth 1.0a* e *OAuth 2.0* si palesa nelle modalità in cui avviene tale validazione. Infatti, mentre nel caso della versione 2.0, dovendo la comunicazione avvenire necessariamente su `SSL`, la chiave segreta viene validata direttamente dal *server*, nel caso della versione 1.0, tale chiave viene firmata sia dal *client* sia dal *server* (con varie complicazioni quali l'ordine degli argomenti). Quindi il protocollo *OAuth 2.0* semplifica i passi 1, 2 e 5 precedentemente illustrati poiché, essendo implementato su `SSL`, elimina il bisogno che *client* e *server* accedano anch'essi ai servizi forniti dal protocollo *OAuth*.

A questo punto il *server* possiede un *access token* equivalente alla coppia *username* e *password* dell'utente. Esso potrà quindi effettuare richieste al *content provider* da parte dell'utente passando tale *access token* come parte della richiesta (e. g., come parametri di query, nell'intestazione `HTTP` o nei dati associati a una richiesta `POST`).

Se il *content provider* può essere contattato solo tramite `SSL`, allora l'implementazione *OAuth* è completa. In caso contrario, invece, è necessario prevedere dei meccanismi di protezione dell'*access token*.

Spesso quest'ultimo problema è risolto con l'utilizzo di un nuovo *access token* (chiamato *refresh token*), equivalente di una password permanente, utilizzato solo per ottenere in cambio degli *access token* con scadenza temporale. Tale approccio, comunque, è utilizzato solo per fornire maggiore sicurezza nel caso in cui l'accesso effettuato non sia criptato tramite connessione `SSL`.

Infine, la fig. 1 mostra il flusso di *OAuth 2.0*.

[7] Tipologia di comunicazione caratterizzata dal fatto di avvenire al di fuori del protocollo principale. In questo contesto, con tale termine, ci si riferisce a qualsiasi richiesta che avviene al di fuori della comunicazione fra il *client* `HTTP` che si sta registrando e il *server* `HTTP` che sta autenticando le credenziali *OAuth*.

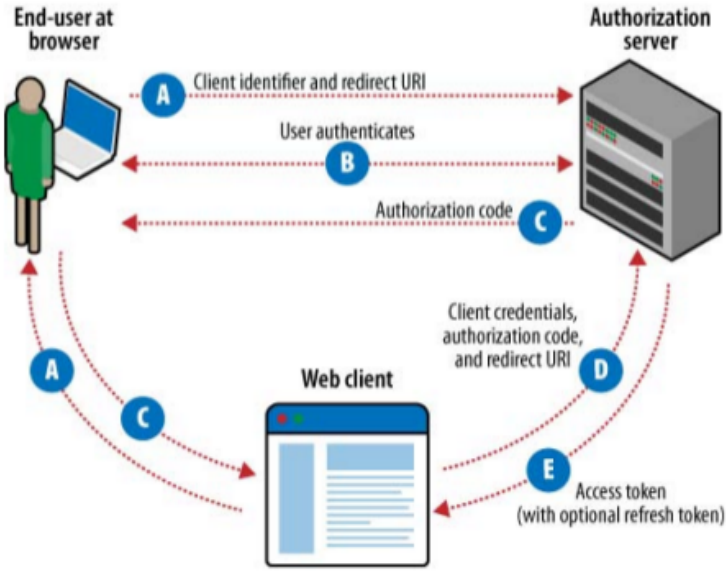

Figura 1: Il flusso del framework *OAuth 2.0*

In conclusione, aldilà dei tecnicismi implementativi, il protocollo *OAuth 2.0* semplifica tutte le comunicazioni fra *client*, *server* e *content provider*. Infatti, dal punto di vista implementativo, il vantaggio principale è la sua ridotta complessità: tale protocollo non richiede procedure di registrazione, riduce la quantità di lavoro necessario ad agire come *client* di un servizio e riduce la complessità della comunicazione tra *server* e *content provider* (permettendo perciò una maggiore scalabilità).

Inoltre, esso incorpora e formalizza alcune estensioni comunemente utilizzate del protocollo *OAuth 1.0a*.

È possibile ottenere maggiori delucidazioni e dettagli tecnici riguardo *OAuth 2.0* all'indirizzo http://oauth.net/2.

2 Facebook Graph API

Al centro di Facebook c'è il *social graph*: un grafo i cui nodi rappresentano le **entità** (i. e., persone, pagine, applicazioni) e i cui archi rappresentano le **connessioni** di tali entità. Qualsiasi entità, o **oggetto**, che opera su Facebook è un nodo di tale grafo sociale. Ogni **azione** che un'entità compie su tale piattaforma identifica un arco (etichettato) uscente dal nodo ad esso correlata. L'etichetta di tale arco è comunemente chiamata **verbo**.

Esiste una sola modalità[8] di interazione con tale grafo sociale, cioè tramite chiamate `HTTP` alle `API` di Facebook. L'interazione è divisa in due componenti.

[8] Ad onor del vero esiste un'ulteriore modalità di interazione: `FQL`. Tuttavia, oltre ad essere obsoleta, tale modalità consiste semplicemente in un dialetto `SQL` che viene compilato in chiamate alle `API` di Facebook. Maggiori informazioni sono reperibili al seguente `URL`: https://developers.facebook.com/docs/reference/fql.

Graph API `API REST` per la lettura e scrittura del grafo sociale.

Open Graph protocol (OGP) Meccanismo che permette di inserire qualsiasi oggetto (e. g., pagina *Web*) nel grafo sociale di Facebook semplicemente inserendo in esso dei meta-dati `RDFa`. Tale piattaforma fornisce anche un insieme di `API HTTP` per l'estensione del grafo sociale anche dal punto di vista delle connessioni che esso supporta.

Per rimarcare l'estensibilità del grafo sociale tramite le *Object API* (permettono la creazione di nuovi verbi e oggetti, previa approvazione), Facebook si riferisce ad esso anche utilizzando il termine *Open Graph*.

Tuttavia, poiché lo scopo di questo lavoro consiste nel presentare le modalità di estrazione dei dati dai *Social Network* si tralascia la trattazione delle `API` e degli strumenti correlati al *Open Graph* e si procede presentando le *Graph API*.

2.1 Introduzione

Le *Graph API* rappresentano un metodo consistente di ottenere una vista uniforme sul grafo sociale di Facebook attraverso delle semplici chiamate `HTTP`. Esse, infatti, permettono di ottenere un sottoinsieme di nodi di tale grafo (e. g., i profili, le foto, gli eventi) e le connessioni che intercorrono tra essi (e. g., le relazioni di amicizia, i contenuti condivisi, e i *tag* nelle foto). Costituiscono perciò il metodo di lettura e scrittura di dati da Facebook. Su di esse è basato tutto il funzionamento di Facebook stesso.

È bene quindi comprendere che ogni qual volta voi creiate un post su Facebook non state semplicemente creando del testo che qualcun altro possa leggere, bensì, state creando un insieme di relazioni strutturate fra nodi (i cui tipi sono predefiniti) del grafo sociale. La fig. 2 evidenzia i concetti espressi.

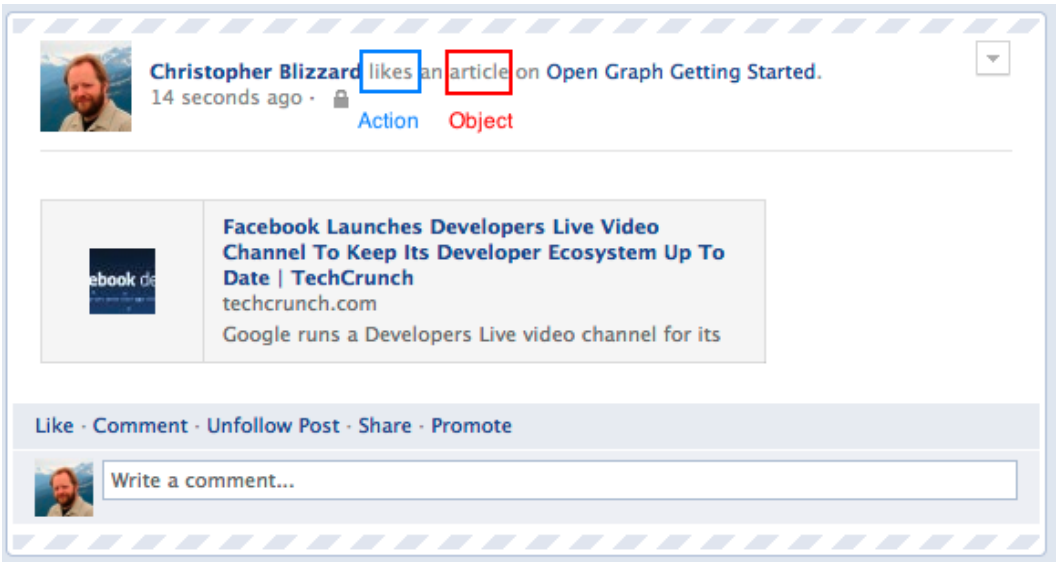

Figura 2: Azioni e oggetti del *Open Graph*

Ogni entità del grafo sociale ha un identificativo univoco, tramite cui esso è indirizzato. Perciò, per accedere alle proprietà di un oggetto è necessario contattare l'indirizzo `https://graph.facebook.com/<id>`. Ad esempio, la pagina ufficiale della *Facebook Platform* ha come identificativo 19292868552, quindi si possono recuperare le proprietà di tale oggetto all'indirizzo https://graph.facebook.com/19292868552, il quale mostrerà il seguente risultato:


```
{
    "about": "Grow your app with Facebook ...",
    "category": "Product/service",
    "company_overview": "Facebook Platform enables anyone to ...",
    "is_published": true,
    "talking_about_count": 42443,
    "username": "FacebookDevelopers",
    "website": "http://developers.facebook.com",
    "were_here_count": 0,
    "id": "19292868552",
    "name": "Facebook Developers",
    "link": "https://www.facebook.com/FacebookDevelopers",
    "likes": 1853490,
    "cover": {
        "cover_id": "10151121467948553",
        "source": "http://sphotos-f.ak.fbcdn.net/hphotos-ak-prn ...",
        "offset_y": 0,
        "offset_x": 0
    }
}
```


In alternativa, tutte le entità con un campo `username` (i. e. profili utente e pagine) sono accessibili usando tale campo in sostituzione dell'identificatore `id`. Perciò, essendo "platform" lo `username` della pagina precedente, contattando l'indirizzo https://graph.facebook.com/platform si otterrà il medesimo risultato.

Si osservi che tutte le risposte sono degli oggetti `JSON`.

Tutti gli oggetti in Facebook sono accessibili coerentemente allo stesso modo. Si presentano di seguito alcuni esempi:

- profili utente: https://graph.facebook.com/giuseppe.vizzari – Giuseppe Vizzari
- pagine: https://graph.facebook.com/cocacola – Coca-Cola
- gruppi: https://graph.facebook.com/195466193802264 – Facebook Developers
- applicazioni: https://graph.facebook.com/2439131959 – Graffiti Wall
- foto: https://graph.facebook.com/98423808305 – Foto dalla pagina Facebook della Coca-Cola

- album di foto: https://graph.facebook.com/99394368305 – Foto pubblicata sulla *timeline*[9] della pagina Facebook della Coca-Cola
- immagini del profilo: http://graph.facebook.com/giuseppe.vizzari/picture – Immagine profilo dell'utente Giuseppe Vizzari[10]

Gli esempi presentati sono relativi solo ad alcune entità del grafo sociale, specificatamente quelle che permettono di recuperare delle informazioni ad esse correlate senza l'utilizzo di alcun tipo di autenticazione e/o autorizzazione (argomento trattato nella sezione 2.2).

Inoltre, tutti gli oggetti del grafo sociale, come detto, sono connessi ad altri. È possibile esaminare le connessioni tra oggetti usando la seguente struttura dell'`URL`:

```
https://graph.facebook.com/<id>/<connection_type>
```

Le connessioni accessibili dal proprio profilo includono:

- lista degli amici: https://graph.facebook.com/me/friends?access_token=..[11]
- news feed[12]: https://graph.facebook.com/me/home?access_token=..
- timeline: https://graph.facebook.com/me/feed?access_token=..
- mi piace: https://graph.facebook.com/me/likes?access_token=..
- mi piace su entità di tipo *Movie*: https://graph.facebook.com/me/movies?access_token=..
- mi piace su entità di tipo *Musician*: https://graph.facebook.com/me/music?access_token=..
- mi piace su entità di tipo *Book*: https://graph.facebook.com/me/books?access_token=..
- note: https://graph.facebook.com/me/notes?access_token=..
- tag nelle foto: https://graph.facebook.com/me/photos?access_token=..
- album di foto: https://graph.facebook.com/me/albums?access_token=..
- tag nei video: https://graph.facebook.com/me/videos?access_token=..
- lista degli eventi a cui si partecipa: https://graph.facebook.com/me/events?access_token=..

[9] La *timeline* è il flusso di oggetti pubblicati su un profilo (e. g., profilo utente, pagina, gruppo o applicazione Facebook). Le foto pubblicate sulla *timeline* vengono automaticamente aggiunte ad un album chiamato "Foto del diario".

[10] Recuperare l'immagine di profilo di un utente in questo modo implica un redirect automatico. Per ottenere invece l'`URL` a cui risiede la foto si utilizzi l'approccio tramite query, i. e. contattando l'indirizzo http://graph.facebook.com/giuseppe.vizzari?fields=picture

[11] Il campo `me` è un identificatore speciale. Maggiori delucidazioni nella sezione 2.3.2.

[12] Il *News Feed* è l'insieme dei post visibili nella home Facebook dell'utente.

- lista dei gruppi cui si è iscritti:

  https://graph.facebook.com/me/groups?access_token=..

- posti: https://graph.facebook.com/me/locations?access_token=..

Si osservi che gli `URL` di richiesta appena mostrati includono il parametro `access_token`, il quale riguarda il processo di accesso autenticato e autorizzato alle informazioni contenute dal grafo sociale di Facebook (argomento trattato nella sezione 2.2).

Sono supportati diversi tipi di connessioni per oggetti diversi. Per esempio, è possibile ottenere la lista delle persone che hanno partecipato (o parteciperanno, nel caso di eventi futuri) all'evento "Le Web Paris 2013" (`id` 151174191736472) recuperando l'elenco all'indirizzo:

https://graph.facebook.com/151174191736472/attending?access_token=...

Tutti i tipi di oggetti e di connessioni supportate sono inclusi nella documentazione[13] delle *Graph API*, alle quali si rimanda per una trattazione dettagliata e esaustiva.

Comunque, è possibile determinare i campi e le connessioni che ogni tipo di oggetto supporta a *run-time*. Per fa ciò è necessario appendere la query `metadata=1` al campo `id` di un oggetto del tipo in questione. Ad esempio, per scoprire i campi e le connessioni supportate dagli eventi è sufficiente collegarsi all'indirizzo:

https://graph.facebook.com/151174191736472?metadata=1&access_token=...

Di seguito un estratto dell'oggetto `JSON` ottenuto per la chiamata `metadata` su un entità di tipo *Event*.

```
...
"metadata": {
  "connections": {
    "feed": "../151174191736472/feed",
    "invited": "../151174191736472/invited",
    "attending": "../151174191736472/attending",
    "maybe": "../151174191736472/maybe",
    "noreply": "../151174191736472/noreply",
    "declined": "../151174191736472/declined",
    "picture": "../151174191736472/picture",
    "admins": "../151174191736472/admins"
  },
  "fields": [
    { "name": "id", "description": ".." },
    { "name": "owner", "description": ".." },
    { "name": "name", "description": ".." },
    { "name": "description", "description": ".." },
    { "name": "start_time", "description": ".." },
```

[13] https://developers.facebook.com/docs/reference/api

```
    { "name": "end_time", "description": ".." },
    { "name": "location", "description": ".." },
    { "name": "venue", "description": ".." },
    { "name": "privacy", "description": ".." },
    { "name": "updated_time", "description": ".." },
    { "name": "picture", "description": ".." },
    { "name": "ticket_uri", "description": ".." }
  ],
  "type": "event"
}
...
```

2.2 Accesso ai dati

Come visto, le *Graph API* consentono di accedere facilmente a tutte le **informazioni pubbliche** di un oggetto. Tuttavia, per ottenere informazioni aggiuntive è invece necessario ottenere l'autorizzazione da parte dell'entità a cui esse appartengono.

In altre parole, le *Graph API* non permettono l'estrapolazione di una immagine completa del grafo sociale di Facebook.

Il meccanismo di autenticazione di Facebook è basato sul protocollo OAuth 2.0, il ché implica la necessità di acquisire un *access token*[14]. Facebook fornisce varie modalità per l'acquisizione dei vari tipi di *access token* di cui dispone. Il metodo più semplice per ottenere un *access token* è accedere al *Graph API Explorer*[15] e premere il pulsante "Get Access Token". Il passo successivo consiste nella selezione dei permessi a cui si è interessati selezionando le rispettive caselle, come mostrato dalla fig. 3.

Chiaramente questa operazione è eseguibile anche tramite `HTTP`, qualora si sia già in possesso di un *access token* valido, semplicemente appendendo alla stringa di interrogazione (i. e., anche detta *query string*) dell'indirizzo `HTTP` delle `API` il parametro `access_token`. Ad esempio:

https://graph.facebook.com/me?access_token=....

Si osservi che, al fine di garantire la privacy dei propri utenti, Facebook impone per quasi tutte le azioni (e quindi per quasi tutte le chiamate possibili alle sue `API`) l'utilizzo di un *access token* che identifichi i permessi richiesti, l'entità che accorda tali permessi e l'identità dell'entità chiamante.

Ad esempio, questo è ciò che accade anche quando si accede su un sito terzo tramite le nostre credenziali Facebook: nel momento in cui si effettua l'accesso, l'applicazioni di autenticazione Facebook del sito terzo richiederà il permesso di accedere ad alcune informazioni del nostro profilo Facebook (operazione che solitamente avviene tramite

[14] Un *access token*, in generale, è una stringa casuale il cui scopo è identificare la sessione associata a una entità a cui sono accordati i permessi per un determinato insieme di azioni. Tale sessione contiene anche l'informazione relativa alla sua durata e alla sorgente che ha richiesto la sua generazione.

[15] https://developers.facebook.com/tools/explorer

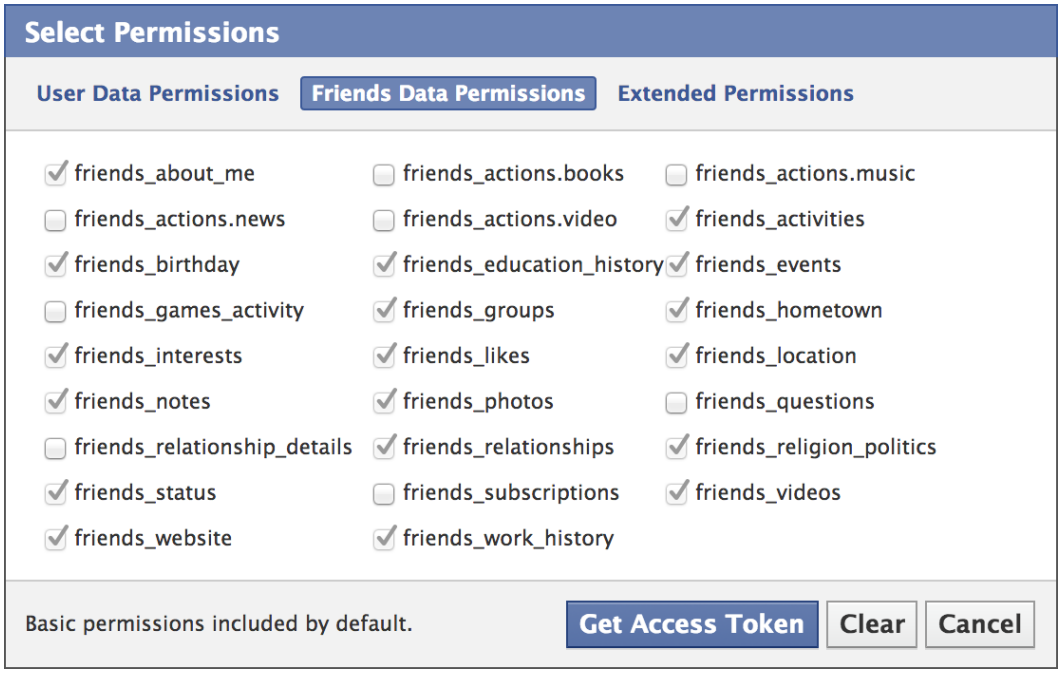

Figura 3: La finestra di selezione dei permessi del *Graph API Explorer*

una finestra *pop-up*), ottenendo e registrando così i permessi necessari (sotto forma di *access token*).

Le *Graph API* prevedono vari tipi di permessi, catalogabili come segue.

1. I permessi base, abilitati di default. La lettura dei dati che richiedono permessi base può quindi essere effettuata senza l'utilizzo di alcun *access token*. Per una trattazione maggiormente esaustiva di tale argomento si rimanda alla sezione 2.4.1.

2. I permessi *user data* e *friends data* sono un insieme di permessi pensati per restringere l'accesso ai dati personali degli utenti. Essi sono distinti in base al target delle richieste `API` (profilo utente dell'entità accordante i permessi o profilo utente dei suoi amici). Un elenco completo e aggiornato di tali permessi è disponibile nella documentazione ufficiale[16] di Facebook.

3. I permessi *extended*, necessari oltre che per la pubblicazione, anche per l'accesso a dati ritenuti altamente sensibili, quali ad esempio il campo `email` di un profilo utente o lo storico degli impieghi ad esso associato. Tali attributi sensibili e i rispettivi permessi sono presentati nella relativa documentazione ufficiale[17].

In conclusione quindi, gli *access token* sono un meccanismo il cui obiettivo è fornire un accesso temporaneo e limitato alle `API` di Facebook.

[16] https://developers.facebook.com/docs/reference/login/extended-profile-properties
[17] https://developers.facebook.com/docs/reference/login/extended-permissions

Per la generazione programmatica degli *access token*, così come per i vari tipi di *access token* di cui Facebook dispone si rimanda alla relativa documentazione ufficiale[18].

2.3 Selezione dei risultati

Le *Graph API* forniscono molte modalità di chiamate finalizzate alla personalizzazione dei risultati. In questa sezione si presentano le modalità principali e più significative.

Per un elenco e una breve descrizione di tutti le modalità è possibile consultare la relativa pagina[19] della documentazione ufficiale Facebook.

2.3.1 Selezione dei campi

Quando si effettua una chiamata alle *Graph API* esse restituiscono, di default, la maggior parte delle proprietà dell'oggetto del grafo sociale correlato alla query sottomessa.

È possibile scegliere quali campi si desidera vengano restituiti utilizzando il parametro `fields` nella stringa di ricerca della chiamata `HTTP` alle `API`. Ad esempio, qualora si desideri solo il l'`id`, il nome (i. e., campo `first_name`) e l'immagine profilo (i. e., campo `picture`) dell'utente "leodido" è possibile contattare l'indirizzo:

https://graph.facebook.com/leodido?fields=id,first_name,picture.

2.3.2 Selezione di oggetti multipli

È possibile richiedere in una singola chiamata alle `API` informazioni relative a più oggetti specificando i relativi identificatori tramite il parametro `ids`. Ad esempio, http://graph.facebook.com/?ids=giuseppe.vizzari,leodido restituirà, nello stesso oggetto `JSON` di risposta, le informazioni di base relative a entrambi i profili degli utenti "giuseppe.vizzari" e "leodido".

Il parametro `ids`, inoltre, accetta anche `URL` come valori. Ciò è utile alla ricerca degli identificatori associati agli `URL` nel *Open Graph*. Ad esempio:

https://graph.facebook.com/?ids=http://www.imdb.com/title/tt0117500/.

restituirà informazioni relative al film "The Rock" (1996) ricavandole dal grafo sociale.

```
<html
    xmlns:og="http://ogp.me/ns#"
    xmlns:fb="http://www.facebook.com/2008/fbml">
<head>
...
```

[18] https://developers.facebook.com/docs/facebook-login/access-tokens
[19] https://developers.facebook.com/docs/reference/api/request-parameters

```
<meta property='og:image' content="http://ia.media-imdb.com/im.."/>
<meta property='og:type' content="video.movie"/>
<meta property='fb:app_id' content='115109575169727'/>
<meta property='og:title' content="The Rock (1996)"/>
<meta property='og:site_name' content='IMDb'/>
...
```

Si osservi, per completezza, che il sorgente `HTML` della pagina http://www.imdb.com/title/tt0117500 appena mostrato definisce un oggetto *Open Graph* tramite l'ontologia `RDFa` di Facebook, specificandone gli attributi con i tag `HTML` `meta`. L'applicazione Facebook del sito IMDB (con `id` 115109575169727) si interfaccia con le *Open Graph API* per inserire i suoi oggetti nel grafo sociale di Facebook.

2.3.3 Identificatore speciale dell'utente corrente

Come già accennato, è possibile utilizzare lo speciale identificatore `me` per riferirsi all'utente corrente. Pertanto, l'indirizzo https://graph.facebook.com/me restituisce il profilo dell'utente attivo.

Tuttavia è doveroso far notare che tale identificatore non consiste in un vero e proprio alias per i percorsi `id` e `username` dell'utente corrente. L'identificatore `me`, infatti, richiede sempre e in ogni caso l'utilizzo di un *access token*.

2.3.4 Selezione dei post con geo-tag

Quando si intende recuperare i post dal *news feed* o dalla *timeline* di un utente, è possibile restringere l'insieme dei risultati al sotto-insieme dei post con geo-tag. A tale fine le *Graph API* forniscono il parametro `with=location`. Ad esempio, la seguente chiamata alle `API` restituirà solo i post nella home (i. e., *news feed*) dell'utente corrente che posseggono un geo-tag, cioè una individuazione geografica.

https://graph.facebook.com/me/home?with=location.

Si noti, comunque, che esiste un tipo di connessione (i. e., `locations`) che permette, previa autorizzazione (cioè tramite *access token*), di accedere ai post geo-taggati di un utente. A tal riguardo si fornisce un esempio (che si consiglia di affrontare in seguito) nell'appendice B.1.

2.4 Estrarre le informazioni di un utente

Prima di proseguire la discussione ulteriore sulle *Graph API* è bene sperimentare i concetti presentati. A tal fine si procede presentando delle porzioni di codice `R` il cui scopo è estrarre i dati relativi a un profilo utente.

2.4.1 Recuperare informazioni di base

Come detto, le *Graph API* prevedono che una serie di campi siano considerati di *default*. Ciò significa che i relativi permessi sono sempre abilitati (anche in mancanza di *access token*) per qualsiasi chiamata alle `API`. Un attento osservatore avrà già notato la dicitura in basso a sinistra nella fig. 3, la quale si riferisce esattamente ai seguenti campi: `id`, `name`, `first_name`, `middle_name`, `last_name`, `gender`, `locale`, `link`, `username`, `picture`.

Tali campi sono quindi reperibili per **qualsiasi utente**. Si osservi, tuttavia, che il campo `picture` fa eccezione, in quanto, anche non richiedendo un permesso apposito, esso viene restituito dalle *Graph API* solamente qualora venga esplicitato nella chiamata alle `API`.

La chiamata alle *Graph API* che si intende effettuare è del tipo:

`https://graph.facebook.com/<username>?fields=<field_1>,...,<field_N>`.

La porzione seguente di codice effettua tale chiamata utilizzando la funzione `facebook`, scritta appositamente per contattare le *Graph API* in `R` e riportata nell'appendice A.1.

```
base_infos <- facebook("leodido")
print(base_infos)
```

```
# $id
# [1] "1461570303"
#
# $name
# [1] "Leonardo Di Donato"
#
# $first_name
# [1] "Leonardo"
#
# $last_name
# [1] "Di Donato"
#
# $link
# [1] "http://www.facebook.com/leodido"
#
# $username
# [1] "leodido"
#
# $gender
# [1] "male"
#
# $locale
# [1] "it_IT"
```

```
fields <- c("id", "name", "first_name", "middle_name", "last_name", "gender", "locale",
    "link", "username", "picture")
# chiamata base con campi esplicitati
field_infos <- facebook("leodido", opts = list(fields = fields))
print(field_infos)
```

```
# $id
# [1] "1461570303"
#
```

```
# $name
# [1] "Leonardo Di Donato"
#
# $first_name
# [1] "Leonardo"
#
# $last_name
# [1] "Di Donato"
#
# $gender
# [1] "male"
#
# $locale
# [1] "it_IT"
#
# $link
# [1] "http://www.facebook.com/leodido"
#
# $username
# [1] "leodido"
#
# $picture
# $picture$data
# $picture$data$url
# [1] "http://profile.ak.fbcdn.net/hprofile-ak-prn1/623721_1461570303_27944717_q.jpg"
#
# $picture$data$is_silhouette
# [1] FALSE
```

2.4.2 Recuperare le informazioni aggiuntive

Come specificato nella sezione 2.2, affinché sia possibile recuperare dati aggiuntivi in modo programmatico da Facebook tramite le sue `API`, è necessario ottenere un insieme di permessi codificati in un *access token*.

Perciò, ipotizzando di aver ottenuto un *access token* abilitando l'opzione `friend_likes` nella sezione dedicata ai permessi *friends data* (i. e., si veda il tab "Friends Data Permissions" nella fig. 3), è quindi possibile accedere ai *like* della nostra cerchia di amici.

Di seguito si mostra come ottenere un determinato numero di *like* da un precisato numero di amici di un utente ("leodido", in questo caso).

```
# token <- '...'
nfriends <- 3
nlikes <- 5
fields <- sprintf("id,name,friends.limit(%d).fields(likes.limit(%d))", nfriends, nlikes)
resp <- facebook("leodido", token = token, opts = list(fields = fields))
resp$id
```

```
# [1] "1461570303"
```

```
resp$name
```

```
# [1] "Leonardo Di Donato"
```

```
for (i in seq.int(nfriends)) {
    ifriend <- resp$friends$data[[i]]
    id <- ifriend$id
    likes <- "not availables"
    if (!is.null(ifriend$likes)) {
        nms <- lapply(resp$friends$data[[i]]$likes$data, "[[", "name")
        cts <- lapply(resp$friends$data[[i]]$likes$data, "[[", "category")
        like_ls <- lapply(seq.int(length(nms)), function(i) sprintf("\n%s (%s)", nms[[i]],
            cts[[i]]))
        likes <- paste(like_ls, collapse = "")
    }
    cat(sprintf("id: %s\nlikes: %s\n\n", id, likes))
}
```

```
# id: 502678298
# likes: not availables
#
# id: 520001802
# likes:
# The Food Box (Food/beverages)
# Nietzsche para Ilustrarse. (Book)
# Revista Literaria La Noche de las Letras (Community)
# NEON RUN MTY (Organization)
# Café Punta del Cielo Monterrey (Restaurant/cafe)
#
# id: 524729879
# likes: not availables
```

Si osservi che l'output dell'esempio di estrazione di dati appena mostrato mostra solamente un sottoinsieme dei dati restituiti dalle *Graph API*. Al fine di presentare un risultato comprensibile, infatti, sono state filtrate (ed eliminate) informazioni quali l'`id` e la data di creazione (i. e., `created_time`) dell'oggetto su cui gli amici dell'utente hanno effettuato *like*; così come sono state omesse le informazioni riguardandi la paginazione (i. e., campo `paging` dell'oggetto `JSON`), utili ad una successiva ed eventuale navigazione (con conseguente esecuzione della stessa query) della lista di amici dell'utente "leodido".

Tuttavia, è anche evidente come i *like* di alcuni amici non siano stati recuperati. Ciò dipende dalle impostazioni di privacy di tali utenti, i quali hanno, con ogni probabilità, ristretto la visibilità relativa alle proprie azioni su Facebook. Di conseguenza, per superare questa limitazione è necessario, come si discute in seguito, complementare l'utilizzo delle *Graph API* con tecnice di *Web Scraping* avanzate.

La query appena effettuata programmaticamente può chiaramente essere eseguita manualmente tramite il *Graph API Explorer*, come mostra la fig. 4.

2.5 Prestazioni

L'esempio mostrato nella sezione 2.4.2 presenta delle particolarità.

Nello specifico esso effettua una chiamata alle API diversa dalle tipologie di chiamate incontrate fin'ora. Perciò, in questa sezione si descrive questa tipologia di chiamate alle *Graph API*, relative all'estrazione in modo efficiente di più informazioni collegate a diverse entità.

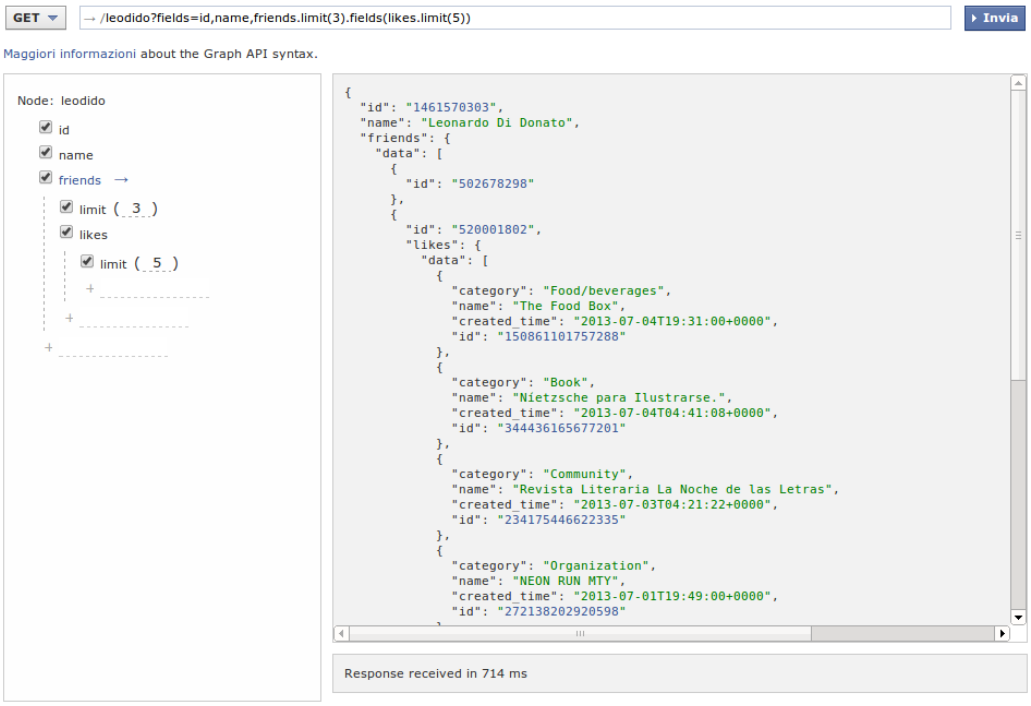

Figura 4: Ottenere i *like* degli amici di un utente Facebook

2.5.1 Espansione dei campi

Si prenda quindi in considerazione l'esempio precedente (sezione 2.4.2).

A tal riguardo si osservi che l'`URL` utilizzato per la chiamata alle *Graph API* presenta una struttura più complicata rispetto a quella basilare composta esclusivamente dalla lista di campi che si desidera ottenere:

```
/leodido/fields=id,name,friends.limit(3).fields(likes.limit(5))&acc..
```

Il campo `friends` è espanso tramite il modificatore `.limit()`, utile a limitare il numero di amici che si intende recuperare. Inoltre, poiché per ognuno di tali oggetti (i. e., amici) si desidera recuperare solo un certo numero `likes` si utilizza il selettore `.fields()` e il modificatore `.limit()` per comporre la query annidata.

Tale funzionalità delle *Graph API*, chiamata appunto **espansione dei campi**, risulta quindi utile a vari fini. Se ne elencano di seguito i principali.

1. Permette di evitare le situazioni di stallo dovute alla richiesta di quantità di dati eccessive, riducendo i campi tramite il limitatore `.limit()`.

2. Permette di effettuare query multiple (e. g., si veda l'esempio precedente), identificabili come query di tipo *join*, in una **singola chiamata** `HTTP`, ottenendo in risposta dei dati strutturati (i. e., sotto-grafo) caratterizzati da una gerarchia di connessioni tra oggetti.

3. Permette di effettuare query annidate fra oggetti del grafo di Facebook, il ché equivale a creare (o recuperare, nel caso sia già definita) una azione del *Open Graph*.

Relativamente a quest'ultima possibilità, si fa notare come i dati ricavati nell'esempio precedente possano essere recuperati anche con una query annidata, perdendo tuttavia le informazioni sul nodo genitore di tipo utente (i. e., utente `leodido`) ma guadagnando in termini di sintesi dei dati. Ecco, di seguito, la porzione di `URL` per effettuare tale chiamata alle `API` tramite una query annidata:

```
/leodido/friends?limit=3&fields=likes.limit(5)&access_token=..
```

Il modo migliore per apprendere questa funzionalità delle *Graph API* è utilizzare il *Graph API Explorer*, il cui menù per la selezione dei campi e degli oggetti guida alla creazione intuitiva di query multiple o annidate. Chiaramente non va tralasciata una lettura esaustiva della relativa documentazione ufficiale[20].

2.5.2 Batch

Qualora sia necessario accedere a quantità significative di informazioni, Facebook, oltre alle query multiple presentate nella precedente sezione, mette a disposizione una tecnica (limitata) di *batching*. Tale tecnica consiste nel combinare le operazioni in un unica chiamata `HTTP` piuttosto che effettuare richieste `HTTP` multiple. Ciò aiuta a ridurre il traffico e i tempi necessari al recupero dei dati.

È possibile recuperare maggiori informazioni nella relativa pagina[21] di documentazione.

2.5.3 Realtime

Le *Graph API* forniscono un meccanismo di aggiornamento in tempo reale di un insieme di informazioni relative ad oggetti e connessioni.

Questa funzionalità permette, ad esempio, di ricevere gli aggiornamenti degli utenti nel momento in cui cambiano alcuni dei suoi dati. Tecnicamente, questa funzionalità è un meccanismo di sottoscrizione: una volta registrata l'entità e/o la connessione che si desidera tenere aggiornata in tempo reale, Facebook comunicherà, tramite una chiamata `HTTP` di tipo `POST`, la lista dei dei cambiamenti avvenuti a un *callback* `URL` specifico[22].

Con tale sottoscrizione, ci si può quindi assicurare che i dati siano aggiornati senza dover necessariamente effettuare la stessa chiamata `HTTP` continuamente alle *Graph API* di Facebook, incrementando così l'attendibilità dei dati, oltre che le prestazioni di estrazione. Con tale metodo di sottoscrizione si evitano, infatti, chiamate alle `API` che restituiscono delle informazioni uguali a quelle restituite dalla chiamata precedente.

[20] https://developers.facebook.com/docs/reference/api/field_expansion
[21] https://developers.facebook.com/docs/reference/api/batch
[22] Necessariamente fornito dall'utente delle *Graph API*.

Si osservi, infine, che non tutti gli oggetti e le connessioni del grafo sociale supportano questa tecnica di aggiornamento dei dati estratti. Per i dettaggli tecnici e implementativi, e per maggiori informazioni relativamente agli oggetti e le connessioni supportate, ci si riferisca alla relativa documentazione ufficiale[23].

2.6 Ricerca

È possibile effettuare ricerche di tutti gli **oggetti pubblici** nel grafo sociale tramite l'indirizzo `https://graph.facebook.com/search`. Il formato è:

`https://graph.facebook.com/search?q=<query>&type=<tipo_oggetto>`

Tutte le chiamate di ricerca alle *Graph API* richiedono che venga passato il parametro `access_token=<token>`. Il tipo di *access token* richiesto dipende, chiaramente, dal tipo di ricerca che si intende effettuare: tutte le ricerche richiedono un *access token* utente eccetto le ricerche su oggetti di tipo pagina (*Page*) o posto (*Place*).

Le ricerche sono supportate per i seguenti tipi di oggetti:

- post pubblici[24]: https://graph.facebook.com/search?q=watermelon&type=post
- profili utente: https://graph.facebook.com/search?q=mark&type=user
- pagine: https://graph.facebook.com/search?q=platform&type=page
- eventi: https://graph.facebook.com/search?q=conference&type=event
- gruppi: https://graph.facebook.com/search?q=programming&type=group
- posti[25]: https://graph.facebook.com/search?q=coffee&type=place&center=37.76,-122.427&distance=1000
- checkin: https://graph.facebook.com/search?type=checkin

È anche possibile cercare nella home (i. e., *news feed*) di uno specifico utente aggiungendo l'argomento `q` all'`URL` per la connessione `home`:

- news feed[26]: https://graph.facebook.com/leodido/home?q=facebook

Al fine di restringere i campi restituiti da una chiamata di ricerca è possibile appendere alla chiamata `HTTP` il parametro `fields` (i. e., `?fields=..`). Ad esempio, per ottenere solo i nomi degli eventi:

[23] https://developers.facebook.com/docs/reference/api/realtime
[24] Nota: questo tipo di ricerca non supporta la paginazione.
[25] Nota: i parametri `center` (richiede due valori: latitudine e longitudine) e `distance` servono a restringere la ricerca; sono tuttavia opzionali.
[26] Nota: richiede che l'utente `leodido` accordi il permesso esteso `read_stream`.

- nomi degli eventi: https://graph.facebook.com/search?fields=name&q=conference&type=event

Tuttavia si osservi che alcuni campi (e. g., `id` e `start_time` nel caso di oggetti di tipo evento) sono sempre restituiti.

Si fa notare, infine, che tale funzionalità delle *Graph API* è, in alcuni casi (e. g., ricerca nel *news feed*) disallineata: restituisce risultati non aggiornati.

Per maggiori delucidazioni su tale funzionalità e sulle sue limitazioni si rimanda alla relativa documentazione ufficiale[27].

2.6.1 Esempio

Di seguito si mostra come utilizzare la funzionalità di ricerca delle *Graph API*.

Si ipotizzi di voler cercare le pagine Facebook relative alle bevande "Pepsi" e "Coke" recuperando anche il numero di *like* così da poter, eventualmente, effettuare delle comparazioni (o aggregazioni, ad esempio) successivamente.

L'esempio riportato si presta a tale scenario.

```
fields <- c("name", "likes")
params <- list(type = "page", fields = fields, limit = 3)
pepsi <- facebook("search", opts = append(params, list(q = "pepsi")))
coke <- facebook("search", opts = append(params, list(q = "coke")))
```

```
print(pepsi$data)
```

```
# [[1]]
# [[1]]$name
# [1] "Pepsi"
#
# [[1]]$likes
# [1] 17113025
#
# [[1]]$id
# [1] "339150749455906"
#
#
# [[2]]
# [[2]]$name
# [1] "Pepsi Brasil"
#
# [[2]]$likes
# [1] 3292232
#
# [[2]]$id
# [1] "112949768466"
#
#
# [[3]]
# [[3]]$name
# [1] "Pepsi Arabia"
#
# [[3]]$likes
```

[27] https://developers.facebook.com/docs/reference/api/search

```
# [1] 1831977
#
# [[3]]$id
# [1] "107173405982788"
```

```
print(coke$data)
```

```
# [[1]]
# [[1]]$name
# [1] "Coca-Cola"
#
# [[1]]$likes
# [1] 69043243
#
# [[1]]$id
# [1] "40796308305"
#
#
# [[2]]
# [[2]]$name
# [1] "Coca-Cola Zero"
#
# [[2]]$likes
# [1] 4566950
#
# [[2]]$id
# [1] "61124008229"
#
#
# [[3]]
# [[3]]$name
# [1] "Coke Studio <U+0628><U+0627><U+0644><U+0639><U+0631><U+0628><U+064A>"
#
# [[3]]$likes
# [1] 3094182
#
# [[3]]$id
# [1] "343815695662228"
```

2.7 Paginazione

Si consideri ancora una volta l'esempio presentato nella sezione 2.4.2, relativo all'estrazione dei *like* degli amici dell'utente "leodido".

Si ipotizzi di volere proseguire l'estrazione dell'esempio in questione recuperando tutti i *like* dei 10 amici successivi dell'utente "leodido". In tal caso è sufficiente modificare leggermente la query (multipla) della chiamata `HTTP` tramite il metodo `.offset()`:

`/leodido/fields=id,name,friends.offset(10).limit(10).fields(likes)`[28]

Questa funzionalità è generalmente chiamata paginazione.

Al fine di filtrare e paginare i dati di una chiamata `HTTP` ci sono diversi parametri che è possibile indicare nelle query delle connessioni:

[28] Si ricordi sempre di aggiungere alle chiamate `API` il parametro contenente l'*access token*: `&access_token=..`

- `limit` e `offset`, ad esempio:

  https://graph.facebook.com/me/likes?limit=3

- `until` e `since` (i. e. un *timestamp unix*[29] o una data accettata dalla funzione PHP `strtotime`[30]), ad esempio:

  https://graph.facebook.com/search?until=yesterday&q=orange.

2.8 Date

Tutti i campi di tipo data sono restituiti come stringhe formattate nello standard ISO-8601. È possibile ottenere le date in un formato diverso specificando il parametro `date_format` nella query (è una modalità di personalizzazione del risultato, si veda a tal proposito la relativa pagina di documentazione ufficiale[31]).

Le stringhe di formato accettate sono identiche a quelle accettate dalla funzione PHP `date`. Ad esempio, l'indirizzo

http://graph.facebook.com/platform/feed?date_format=U.

restituisce il *feed* della pagina "Facebook Platform", con le date formattate come *timestamp unix*.

2.9 Limiti e vincoli

Durante la discussione fin qui affrontata si è già accennato ad alcune limitazioni quantitative riguardanti le informazioni ottenibili tramite le *Graph API*.

Si è già osservato, infatti, che ogni chiamata alle *Graph API* restituisce **solo** gli oggetti la cui **visibilità** è **pubblica**. Le `API` di Facebook forniscono un parametro (chiamato *privacy parameter*[32]) per modificare tale impostazione. Tuttavia questo parametro funziona solo per le chiamate `HTTP` di tipo `POST` (i. e., funziona solo in scrittura).

Inoltre, dall'esempio creato per estrarre i geo-tag dei post della cerchia di amici di un dato utente (si veda l'appendice B.1) è emerso che il numero massimo di post geo-taggati ottenibile per ogni singolo amico è fissato a 25.

Limitazioni simili a questa appena esposta si verificano anche relativamente all'estrazione di altri oggetti. Ad esempio, non è possibile estrarre tutti i post di una pagina Facebook poiché i post più vecchi di una certa data (soglia variabile da pagina a pagina, nè tantomeno comunicata da Facebook) vengono spostati da Facebook su una livello (inteso come base di dati) diverso e non accessibile via `API`.

Questo meccanismo è quello che generalmente impedisce di accedere a tutto lo storico delle informazioni relative a un oggetto del grafo sociale di Facebook tramite le *Graph API*.

[29] http://en.wikipedia.org/wiki/Unix_time
[30] http://php.net/manual/en/function.strtotime.php
[31] https://developers.facebook.com/docs/reference/api/request-parameters/
[32] https://developers.facebook.com/docs/reference/api/privacy-parameter

Inoltre, tale meccanismo, è ciò che spinge gli sviluppatori avari di informazioni e dati (seppur protetti da *privacy*) a utilizzare tecniche di estrazione ben più avanzate (a tal riguardo si veda la sezione 4).

Inoltre, anche dal punto di vista quantitativo, la politica[33] adottata da Facebook è quella di considerare le seguenti soglie come limitanti:

- massimo 600 chiamate alle `API` in 600 secondi per *access token* (i. e., per ogni utente)
- ogni applicazione Facebook può effettuare massimo 100 milioni di chiamate giornaliere alle `API`.

Si osservi, comunque, che tali soglie non costituiscono dei vincoli tecnici. Nel momento in cui vengono oltrepassate, tuttavia, il servizio degraderà velocemente verso la non utilizzabilità. Sarà quindi necessario estendere tali limiti contattando[34] Facebook e contrattando dei nuovi termini di contratto per l'accesso alle *Graph API*.

2.10 Considerazioni finali

In questa sezione si sono presentate le Facebook *Graph API* al fine di permettere al lettore di comprendere appieno il loro funzionamento. Una trattazione esaustiva e maggiormente dettagliata delle `API`, infatti, oltre a essere poco trattabile, risulterebbe poco utile (a tale scopo esiste la documentazione ufficiale[35] poiché esse subiscono spesso delle migliorie o modifiche. Perciò è importante comprendere la struttura grafico centrica delle `API` di Facebook e la sua relazione alla creazione degli `URL` di chiamata. Compreso tale meccanismo è consigliabile sperimentare le *Graph API* (e. g., oggetti, relative connessioni, relativi campi etc. etc.) tramite il *Graph API Explorer*[36].

Qualora si desideri procedere approfondendo l'argomento dal punto di vista pratico, l'appendice B riporta alcuni esempi maggiormente dettagliati. Si fornisce inoltre la pagina ufficiale degli esempi[37] approntata da Facebook.

3 Twitter API v1.1

Twitter è un servizio di *real-time micro-blogging* il cui scopo è permettere agli utenti di inviare brevi messaggi (i.e., composti massimo da 140 caratteri) chiamati *tweet*.

Diversamente da altri *Social Network* (e.g., Facebook) le connessioni non sono bidirezionali: la rete di informazioni sottostante Twitter è una **rete asimmetrica**, composta da amici (i.e., *friend*) e *follower*. Mentre i *friend* sono gli *account* che un

[33] *Facebook Policy*: https://developers.facebook.com/policy
[34] *Contattare Facebook*: https://www.facebook.com/help/contact/?id=206103619499608
[35] *Facebook Developers*: https://developers.facebook.com/docs/reference/api
[36] *Graph API Explorer*: https://developers.facebook.com/tools/explorer
[37] Esempi: https://developers.facebook.com/docs/reference/api/examples

utente segue, i *follower* sono gli *account* che invece seguono l'utente. Ciò denota una differente nozione di amicizia in Twitter, principale differenza rispetto a Facebook.

La *timeline*[38] di un utente è quindi un flusso *real-time* contenente tutti i *tweet* dei suoi *friend*, cioè degli *account* che esso segue.

Twitter offre una vasta collezione di `API`, tutte basate su `HTTP`. Lo scopo di questa sezione è presentarne i concetti e le modalità d'uso principali.

Così come Facebook, Twitter mette a disposizione un'applicazione utile all'esplorazione delle sue `API`: la *Twitter Console*, mostrata dalla fig. 5.

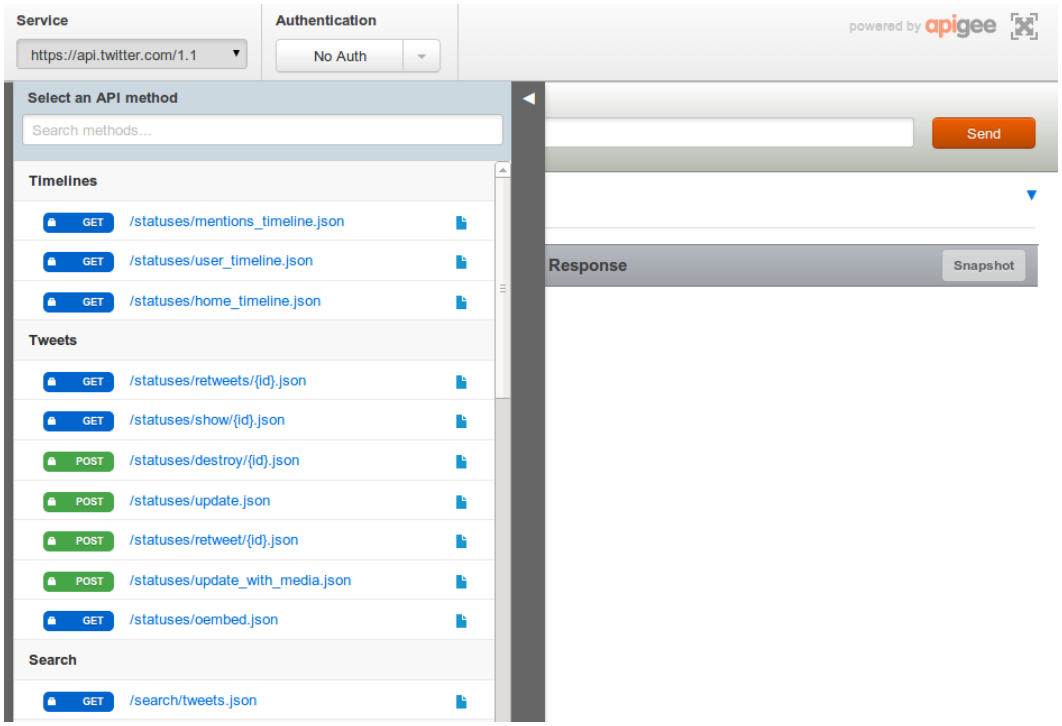

Figura 5: La *Twitter Console*

3.1 Metodi di autenticazione

Twitter supporta vari metodi di autenticazione basati sul protocollo *OAuth*, ognuno dei quali è utile in un determinato contesto applicativo (per alcuni esempi di *mapping* si consulti la relativa documentazione ufficiale[39]).

In prims, è possibile distinguere tra due tipi di autorizzazione: uno per conto degli utenti, basato principalmente (ad eccezione di alcuni casi) su *OAuth 1.0a* e uno per conto di un'applicazione, basato su *OAuth 2.0*.

Si osservi che, a differenza di Facebook, Twitter non prevede un insieme di permessi correlati al tipo di dati che l'utente può o meno accordare. Gli *access token* codificano quindi solamente il permesso di accesso ai dati.

[38] https://support.twitter.com/entries/164083-what-is-a-timeline
[39] https://dev.twitter.com/docs/auth/obtaining-access-tokens

3.1.1 OAuth 1.0a

Come anticipato, Twitter fornisce varie modalità per effettuare richieste autorizzate alle sue `API` per conto di un utente. Ad esempio, esso fornisce un meccanismo di autenticazione basato su `PIN` per le applicazioni *mobile* o in generale per le applicazioni *embedded* (i. e., che non hanno accesso ad un *Web Browser*). Comunque, le varie modalità di autorizzazione[40], aldilà di alcune piccole differenze a seconda del contesto applicativo (e. g., diversi *end-point*), seguono la specifica del protocollo *OAuth 1.0a*. Ciò implica che sarà necessario inviare una richiesta di autorizzazione `HTTP` che comunichi:

1. quale applicazione sta effettuando la richiesta
2. per conto di quale utente l'applicazione sta effettuando la richiesta
3. se l'utente ha autorizzato o meno l'applicazione
4. se durante il transito la richiesta è stata manomessa da terzi

Ognuno di questi requisiti si traduce in uno o più parametri nell'intestazione della richiesta `HTTP`:

- punto 1: `oauth_consumer_key`
- punti 2 e 3: `oauth_token`
- punto 4: `oauth_signature` e `oauth_signature_method`.

Inoltre, tutte le richieste alle Twitter `API` devono contenere altri 3 parametri: `oauth_nonce`, `oauth_timestamp` e `oauth_version`. Maggiori informazioni circa la richiesta di autorizzazione (i. e., circa l'entità di tali parametri) sono disponibili nella relativa documentazione ufficiale[41].

Questa modalità di autenticazione (e autorizzazione) è chiamata *application-user authentication*.

3.1.2 Autenticazione per sole applicazioni

Twitter offre alle applicazioni la possibilità di inviare richieste autenticate per conto di esse stesse (invece che per conto di uno specifico utente). Tale meccanismo di autenticazione e autorizzazione (i. e., implementa un meccanismo di concessione delle credenziali), anche detto *application-only authentication*, è basato sulla specifica di *OAuth 2.0*.

Il flusso di questa modalità è così costituito:

1. l'applicazione codifica la sua *consumer key* e la sua *secret key* in un insieme di credenziali codificato (secondo la specifica RFC 1738[42])

[40] Le varie modalità di autorizzazioni sono documentate all'indirizzo https://dev.twitter.com/docs/auth/obtaining-access-tokens.
[41] https://dev.twitter.com/docs/auth/authorizing-request
[42] http://www.ietf.org/rfc/rfc1738.txt

2. l'applicazione effettua una richiesta `HTTP POST` all'*end-point* `oauth2/token` al fine di scambiare queste credenziali con un *bearer token*.

Si può a questo punto utilizzare il *bearer token* ottenuto al fine di autenticare l'applicazione che esso rappresenta nelle `REST API`, cosicché esse siano utilizzabili (a fini di lettura e/o scrittura di dati in e da Twitter).

La fig. 6 mostra il processo appena descritto.

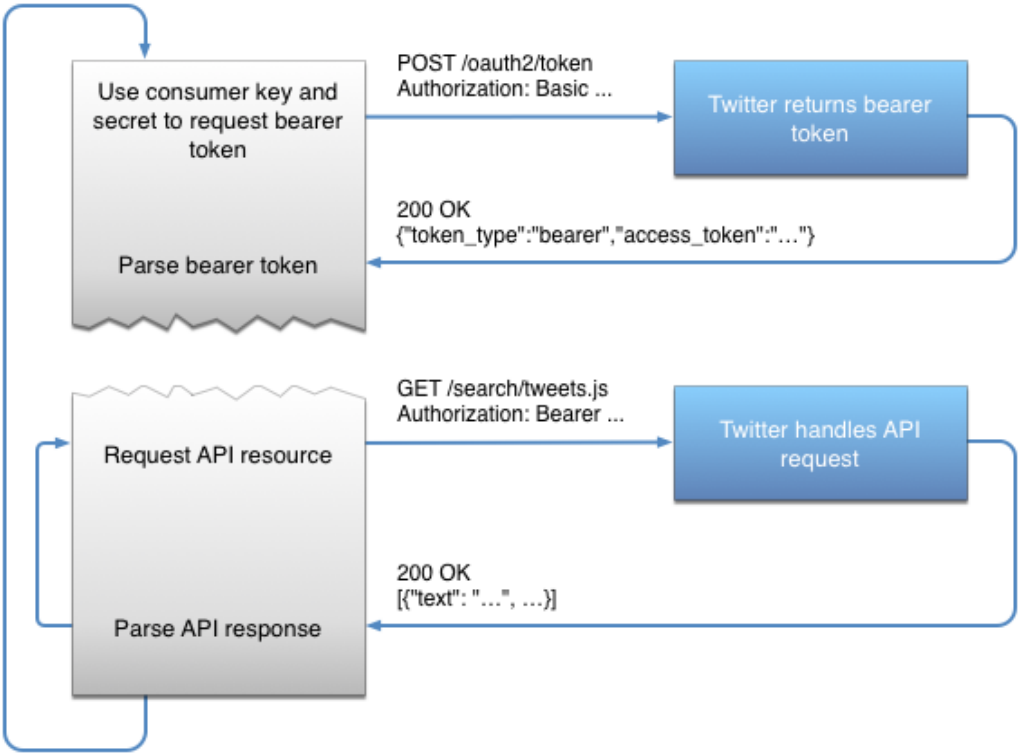

Figura 6: Processo di autenticazione *OAuth 2.0* per sole applicazioni Twitter

Si osservi, come già specificato nella sezione relativa (1.3), che questo approccio è molto più semplice (sia concettualmente, sia dal punto di vista implementativo) rispetto al modello di autorizzazione e autenticazione di *OAuth 1.0a*, poiché non è richiesto che la chiamata sia firmata.

Poiché questo metodo di autorizzazione (e autenticazione) è un'implementazione di *OAuth 2.0*, è assolutamente richiesto che le chiamate verso l'*end-point* siano di tipo `SSL` (i. e., usino cioè il protocollo `HTTPS`).

Inoltre, in tale situazione non esiste alcun contesto relativo all'utente: ne consegue che alcuni *end-point* non funzionano quando si utilizza questo metodo di autenticazione.

Maggiori dettagli implementativi sono reperibili nella documentazione ufficiale[43].

[43] https://dev.twitter.com/docs/auth/application-only-auth

3.1.3 Ottenere una chiave per le API

Le Twitter `API` prevedono che tutte le richieste, ad esclusione di poche eccezioni, forniscano i parametri richiesti al fine di implementare un'autorizzazione *OAuth*. Perciò, è necessario procurarsi i valori di tali parametri. A tale scopo è indispensabile creare un'applicazione Twitter collegandosi al gestore delle applicazioni[44].

Il processo di creazione è banale: richiede semplicemente l'inserimento di un nome, una descrizione e un indirizzo web (anche fittizio) per l'applicazione.

A scopo illustrativo si è creata un'applicazione di test, riportata dalla fig. 7.

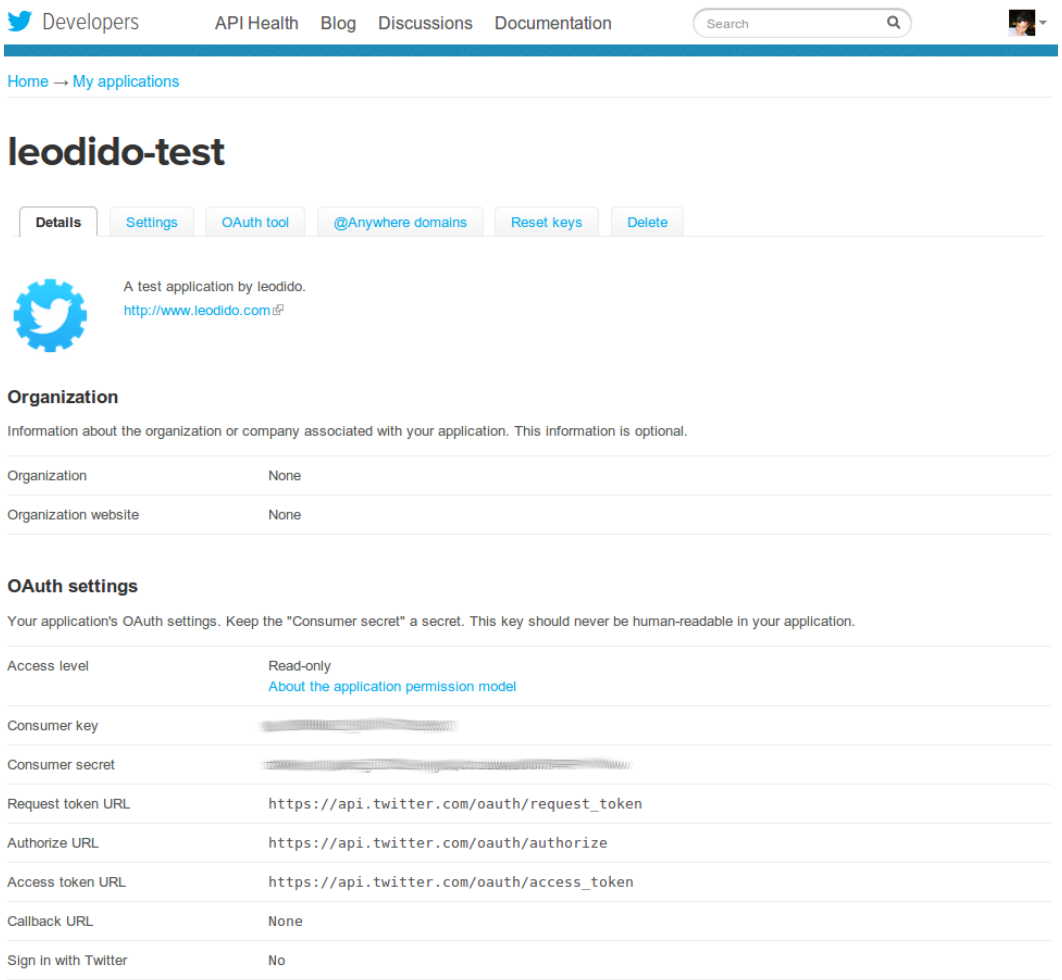

Figura 7: Schermata riportante un'applicazione Twitter

Come visibile dalla fig. 7 si è così ottenuto una *consumer key* e una *secret key* (o *consumer secret*), che si ricorda, sono indispensabili nel processo di autenticazione e autorizzazione delle Twitter `API` in quanto, codificate, identificano univocamente la corrente applicazione.

[44] https://dev.twitter.com/apps

Si osservi inoltre che è possibile specificare i **permessi** conferiti all'applicazione, che sono solo di tre tipi: lettura, scrittura, accesso ai messaggi diretti.

Infine, al fine di facilitare la sperimentazione delle `API`, Twitter fornisce un modo per ottenere in modo semplice un *access token* che rappresenti l'utente creatore (e il relativo contesto) dell'applicazione, cosicché si possano sperimentare le `API` sul proprio *account* prima di implementare qualcuno dei complessi meccanismi di autenticazione forniti nella propria applicazione.

Per ottenere un *access token* associato al proprio *account* è sufficiente cliccare sul pulsante mostrato in fig. 8.

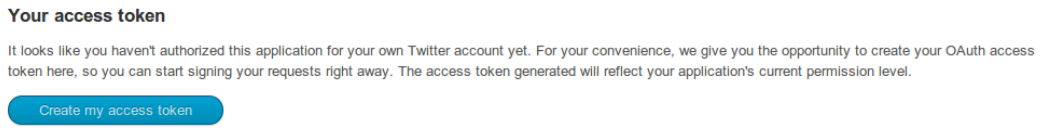

Figura 8: Richiedere un *access token* per il proprio *account* Twitter

Ottenendo, come mostrato dalla fig. 9 l'*access token*[45].

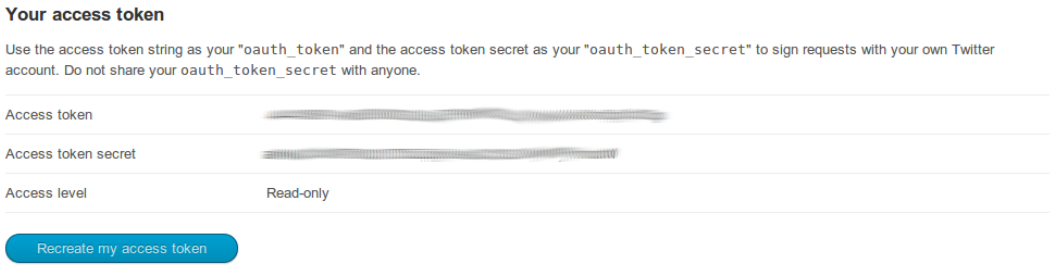

Figura 9: Schermata riportante l'*access token* per il proprio *account* Twitter

Si noti che è possibile evitare i passi descritti in questa sezione utilizzando la *Twitter Console* per generare un *access token*:

- cliccare sul menù a discesa e selezionare l'autenticazione *OAuth 1*
- effettuare l'accesso con le proprie credenziali Twitter.

Così facendo la *Twitter Console* creerà un *access token* (oltre a una *consumer key* e *secret key* per un'applicazione temporanea e fittizia) e permetterà di effettuare delle chiamate alle Twitter `API`. Tuttavia, a differenza del *Facebook Graph Explorer*, la *Twitter Console* non mostra l'*access token*.

Perciò, qualora si vogliano utilizzare le `API` in modo programmatico, è necessario seguire i semplici passi esposti nella corrente sezione.

[45] L'*access token* non è mostato per motivi di sicurezza personale, tanto più che, in questo caso, esso non ha una scadenza prefissata.

3.1.4 Effettuare l'autenticazione

In questa sezione si presenta come effettuare il tipo più semplice di autenticazione per le Twitter `API`, la *application-only authentication*, tramite la funzione `twitter_app_only_auth` presentata e discussa nella sezione A.2.

```
# tw_app_key <- '...'  tw_app_secret <- '...'

twitter_app_only_auth(tw_app_key, tw_app_secret, store = TRUE)
```

```
# To enable the connection, please direct your web browser to:
# https://api.twitter.com/oauth/authorize?oauth_token=YJybDZ7yQyXJnphJSzFVTzK2E2rS3Cyorj5bcNQ9NlE
# When complete, record the PIN given to you and provide it here:


# Error: Unauthorized
```

Come confermato dall'output, questo procedimento di autenticazione richiede l'apertura di un URL (contenente il *bearer token* come parametro, si veda il parametro `oauth_token` della *query string*): la pagina a tale indirizzo richiederà all'utente di autorizzare l'applicazione `leodido-test` e genererà in risposta un `PIN` numerico che l'utente deve manualmente copiare e inserire nel terminale. Una volta compiuta questa operazione, se il parametro `store` della funzione `twitter_app_only_auth` è impostato a `TRUE`, non sarà più necessario ripeterla poiché le credenziali verranno serializzate su disco cosicché siano caricabile ogni qual volta ce ne sia bisogno tramite la funzione `load_twitter_creds` (si veda la sezione A.2 per la sua implementazione).

3.2 Risorse

Così come per ogni altra `REST API`, gli *end-point* delle Twitter `API` sono del tipo:

`https://api.twitter.com/1.1/<resource>/<action>`.

Le Twitter `API` constano di 16 risorse: *timeline*, *tweet*, ricerca, *streaming*, messaggi diretti, *friends* e *follower*, utenti, utente suggerito, favoriti, liste, ricerca salvata, *places*, trends, *spam reporting*, *OAuth*, *help*.

Per una descrizione dettagliata di tali risorse e delle rispettive azioni supportate si rimanda alla relativa documentazione ufficiale[46].

3.2.1 Esempio

A puro titolo dimostrativo si mostra come, avendo già effettuato l'autenticazione *OAuth 2.0* e memorizzato le credenziali su disco, è possibile contattare le Twitter `API` con estrema facilità; anche grazie alla libreria `twitteR` (Gentry 2013) che mette a disposizione un *wrapper* per ogni *end-point* esistente.

```
invisible(load_twitter_creds())
me <- getUser("leodido")
print(me$followersCount)
```

[46] https://dev.twitter.com/docs/api/1.1

```
# [1] 163
```

Per ispezionare i metodi e le proprietà fornite dal pacchetto `twitteR` per ogni risorsa Twitter è possibile usare la funzione `str` (i. e., *structure*) di `R`. Così facendo si apprende, ad esempio, delle proprietà `description`, `name` e `lastStatus` (che è un oggetto caratterizzato a sua volta da un insieme di proprietà e metodi), che visualizziamo di seguito. estrema facilità; anche grazie alla libreria `twitteR` (Gentry 2013) che mette a disposizione un *wrapper* per ogni *end-point* esistente.

```
print(me$name)
```

```
# [1] "Leo Di Donato"
```

```
print(me$description)
```

```
# [1] "Which really tech enthusiast guy has the time to make a real personal website? C'mon .."
```

```
print(me$lastStatus$created)
```

```
# [1] "2013-07-16 13:09:21 UTC"
```

3.3 Ricerca

La *Twitter Search API* è parte integrante della versione 1.1. delle `REST API`. Essa permette l'esecuzione di *query* in tempo reale sui *tweet* **recenti**. È importante tenere presente le seguenti caratteristiche di questa parte delle `API`:

- l'indice in cui viene effettivamente eseguita la *query* sottomessa include dai 6 ai 9 giorni di *tweets*
- i *tweet* più vecchi di una settimana sono rimossi dall'indice
- le *query* devono essere semplici, può infatti accadere che le `API` restituiscano il seguente messaggio di errore

```
{
    "error":
    "Sorry, your query is too complex. Please reduce compl ..."
}
```

- la funzionalità di ricerca fornita si focalizza sulla rilevanza dei risultati, non sulla completezza[47] dell'insieme restituito: ciò implica che alcuni utenti e/o *tweet* possono essere persi e non riportati dai risultati di ricerca
- le *query* sono limitate ad un massimo di 1000 caratteri, operatori inclusi

[47] Nel caso in cui la completezza dei risultati sia richiesta è necessario rivolgersi alle *Twitter Streaming API*.

- è sempre richiesta una qualche forma di autenticazione, *OAuth 1.0a* o *OAuth 2.0* (*application-only authentication*).

Si noti che l'unico *end-point* (e quindi l'unica risorsa) disponibile per tale funzionalità è: `/search/tweets`[48].

Maggiori dettagli (e. g., operatori di ricerca supportati, *best practices*) sono disponibili nella relativa pagina di documentazione ufficiale[49].

3.3.1 Esempio

Lo scopo di questo esempio è fornire una breve introduzione all'utilissimo metodo `searchTwitter` della libreria `twitteR` e al contempo dimostrare come, tale libreria `R`, insieme alla *application-only authentication*, renda notevolmente più semplice l'estrazione di dati dal Twitter, a differenza di quanto invece accade nelle situazioni in cui si utilizza la *application-user authentication* (come mostrato dall'esempio nella sezione B.2).

```
invisible(load_twitter_creds())
results <- searchTwitter("user2013", n = 10)
counter <- 0
invisible(lapply(results, function(t) {
    counter <<- counter + 1
    username <- t$screenName
    date <- t$created
    text <- paste(substring(t$text, 0, 80), "...")
    cat(sprintf("(%d)\n[%s]\n<@%s> tweeted:\n%s.\n\n", counter, date, username, text))
}))
```

```
# (1)
# [2013-07-18 10:52:19]
# <@adolfoalvarez> tweeted:
# RT @ConcejeroPedro: #useR2013 You must Read this about R naming conventions http ....
#
# (2)
# [2013-07-18 10:16:33]
# <@ConcejeroPedro> tweeted:
# #useR2013 You must Read this about R naming conventions http://t.co/wI1hT7kwDT p ....
#
# (3)
# [2013-07-18 09:49:31]
# <@romain_francois> tweeted:
# RT @revodavid: Speedup #rstats code by inlining C++ -- @revojoe's #useR2013 expe ....
#
# (4)
# [2013-07-18 02:12:58]
# <@fellgernon> tweeted:
# RT @revodavid: Speedup #rstats code by inlining C++ -- @revojoe's #useR2013 expe ....
#
# (5)
# [2013-07-17 22:44:28]
# <@eoinbrazil> tweeted:
# RT @revodavid: Speedup #rstats code by inlining C++ -- @revojoe's #useR2013 expe ....
#
```

[48] Documentazione relativa all'*end-point* di ricerca: https://dev.twitter.com/docs/api/1.1/get/search/tweets.

[49] https://dev.twitter.com/docs/using-search

```
# (6)
# [2013-07-17 21:56:25]
# <@divyakkm> tweeted:
# RT @revodavid: Speedup #rstats code by inlining C++ -- @revojoe's #useR2013 expe ....
#
# (7)
# [2013-07-17 21:51:45]
# <@eddelbuettel> tweeted:
# RT @revodavid: Speedup #rstats code by inlining C++ -- @revojoe's #useR2013 expe ....
#
# (8)
# [2013-07-17 21:46:40]
# <@revodavid> tweeted:
# Speedup #rstats code by inlining C++ -- @revojoe's #useR2013 experiences with Rc ....
#
# (9)
# [2013-07-17 20:57:16]
# <@badlogicgames> tweeted:
# Interesting, R on the JVM http://t.co/Aj6rOGyPva ....
#
# (10)
# [2013-07-17 17:01:02]
# <@CornelissenJo> tweeted:
# RT @statslabdublin: http://t.co/EYilMw5Dpu : looks quite interesting useful teac ....
```

3.4 Streaming

Le *Twitter Streaming API* offrono un insieme di funzionalità a bassa latenza utili ad accedere in tempo reale il flusso globale di *tweet*. Questo insieme di funzionalità forniscono quindi, così come fa il meccanismo *realtime* delle *Facebook Graph API* (si veda la sezione 2.5.3), un meccanismo di aggiornamento in tempo reale delle informazioni relative ad determinate risorse, eliminando perciò la necessità di chiamare ripetutamente ad intervalli regolari il relativo *end-point* `REST` (tecnica chiamata *polling*).

Twitter offre alcuni *end-point* di tipo *streaming*. Li si riporta di seguito, fornendone una breve descrizione.

Flusso pubblico Informazioni pubbliche che fluiscono in Twitter. Utile a seguire specifici utenti o argomenti (ottimo per effettuare *data mining*).

Flusso dell'utente Flusso relativo al singolo utente: contiene pressappoco tutti i dati corrispondenti alla vista di un singolo utente sul suo account Twitter.

Flusso degli utenti Versione multi-utente della precedente tipologia di *streaming*. Pensato per i *server* che devono connettersi a Twitter per conto di un gran numero di utenti.

I rispettivi *end-point*, in ordine di presentazione, sono:

1. `statuses/filter` (`POST`), `statuses/sample` e `statuses/firehose` (`GET`)
2. `user`[50] (`GET`)

[50] Si noti che differisce dall'*end-point* `users`.

3. `site` (`GET`).

A differenza di quanto accade per le `REST API` standard, connettersi alle *Twitter Streaming API* richiede l'instaurazione di un connessione `HTTP` persistente. Ciò influisce, chiaramente, su tutta l'architettura dell'applicazione che utilizza tale insieme di funzionalità.

Per interfacciarsi a queste funzionalità tramite `R` è consigliato l'utilizzo del pacchetto `streamR` (Barbera 2013).

È possibile recuperare informazioni aggiuntive circa tale funzionalità consultando la relativa documentazione[51].

3.5 Limiti

Twitter prevede due tipi di limiti per le sue `API`, relativi alle due categorie di autenticazione: *application-user* e *application-only*. Inoltre, queste limitazioni sono indipendenti l'una dall'altra.

I limiti nella versione attuale delle Twitter `API` (ie{}, 1.1) sono considerati primariamente in base all'*access token* (i. e., in base all'utente). Nel caso in cui, invece, si utilizzi la *application-only authentication* tali limiti vengono determinati globalmente per l'intera applicazione.

La **finestra** di validità di ogni richiesta `HTTP` è di 15 **minuti**.

Quindi, a seconda della risorsa che si richiede si determinano dei limiti diversi in base al tipo di autenticazione utilizzata e al rispettivo numero di richieste massimo per finestra (i. e., `RPAUA`[52] o `RPAOA`[53]).

Ad esempio, per i *follower* è possibile chiamare l'azione `ids` (che restituisce una lista contenente gli `id` dei *follower* dell'utente per cui si effettua la richiesta) massimo una volta al minuto (qualsiasi sia il metodo di autenticazione, i.e. `RPAUA = 15`, `RPAUA / 15 = 1`). Nel caso invece si vogliano ricavare le liste dei *follower* dell'utente, è possibile effettuare massimo 2 chiamate al minuto solo nel caso in cui si utilizzi *application-only authentication* (altrimenti massimo 1).

Si noti che, in caso di autenticazione per conto di utenti, il limite di ricerca consiste in 12 *query* al minuto poiché `RPAUA = 180` (i. e., `RPAUA / 15 = 12`). Invece, usando la *application-only authentication* tale limite sale a 30 *query* al minuto poichè `RPAUA = 450`.

Tali informazioni sono comunque ricavabili dalla tabella nella sezione C.1.

Le Twitter `API` supportano un tipo di chiamata `HTTP`,

`GET application/rate_limit_status`,

finalizzata ad apprendere il numero di richieste ancora effettuabili per una determinata risorsa, indipendentemente dal tipo di autenticazione utilizzata.

[51] https://dev.twitter.com/docs/streaming-apis
[52] Numero di richieste per finestra in caso di *application-user authentication*.
[53] Numero di richieste per finestra in caso di *application-only authentication*.

Ad esempio, il risultato di tale chiamata con un *bearer token* (i. e., *application-only authentication*) per la risorsa `search/tweets` è il seguente:

```
{
    "rate_limit_context": {
        "application": "nQx7..."
    },
    "resources": {
        "search": {
            "/search/tweets": {
                "limit": 450,
                "remaining": 420,
                "reset": 1362436375
            }
        }
    }
}
```

Maggiori informazioni sono disponibili nella documentazione ufficiale[54].

3.6 Considerazioni

Gli *account* protetti non sono accessibili tramite `API`.

Termini di contratto: https://dev.twitter.com/terms/api-terms.

Termini di servizio: https://twitter.com/tos.

4 Web Scraping

In questa sezione si discute il principale metodo utile all'estrazione di dati dai siti *Web* senza utilizzare le `API` ufficiali. Nello specifico si affronta l'insieme di metodi cu ci si riferisce generalmente con l'espressione *Web Scraping*.

Il termine *Web Scraping* si riferisce un insieme di tecniche per l'estrazione automatica del contenuto dai siti web su `HTTP`, finalizzate alla trasformazione di tale contenuto in informazione strutturata utilizzabile in altri contesti. Semplificando si può definire il processo di *scraping* come il processo di estrazione dei dati da pagine `HTML`; perciò qualsiasi contenuto *Web* visibile tramite un *browser* può teoricamente essere scaricato dalla rete e processato.

Le fasi principali in cui si compone l'intero processo di estrazione sono le seguenti.

1. Recupero delle pagine `HTML` (i. e., *crawling*)
   1.1 Gestione della paginazione
   1.2 Gestione delle richieste asincrone

[54] https://dev.twitter.com/docs/rate-limiting/1.1

2. Estrazione e strutturazione dei dati (i. e., *scraping*)

Quindi, il primo passo necessario è conoscere l'indirizzo (i. e., *end-point*) cui effettuare la richiesta iniziale.

A tale punto, è necessario approntare un *crawler*[55] che navighi il sito *target* al fine di collezionare le pagine `HTML` desiderate.

Si noti che il processo di *crawling* è regolamentato da uno "standard de-facto": il protocollo di esclusione dei *robot*[56]. Esso consiste in un file `robots.txt` posto nella cartella *home* del sito *Web* (e. g., https://facebook.com/robots.txt) che ha il compito di comunicare ai *crawler* (e. g., a quelli utilizzati dai motori di ricerca), tramite delle regole, i percorsi `HTTP` visitabili e indicizzabili del succitato sito *Web*. Si noti tuttavia che tale protocollo, non essendo integrato nel protocollo `HTTP`, non forza in alcun modo che le sue regole vengano rispettate. Perciò, tale file è solitamente utilizzato al semplice scopo di indicare la politica che un sito adotta relativamente al *crawling* e al *scraping* dei suoi contenuti.

Il processo di *crawling*, precedente a quello di *scraping*, può quindi considerare anche i seguenti passi:

- verificare l'esistenza di una funzionalità di ricerca fornita dal sito *target*, ed in caso affermativo sfruttarla; ad esempio simulando query di ricerca tramite il parametro `q` nella *query string* dell'*end-point* di ricerca
- implementare la funzionalità di paginazione, ad esempio tramite l'utilizzo del parametro `offset` nella *query string* degli `URL`, per recuperare i dati meno recenti
- prendere in considerazione le porzioni di pagina caricate in modo asincrono tramite `AJAX`[57].

Quest'ultimo punto è solitamente la parte più ostica dal punto di vista implementativo poiché è necessario investigare a fondo il sito *target* al fine di individuare le modalità in cui esso utilizza `AJAX` per il caricamento asincrono. Ad esempio, un sito *target* potrebbe rendere visibili le sue richieste `AJAX` appendendo dei frammenti `#` all'`URL` delle pagine o nascondere le sue richieste `AJAX` indirizzandole verso un altro *end-point*. Ad esempio, Facebook indirizza le sue richieste `AJAX` verso un *end-point* `/ajax` (e. g., https://facebook.com/ajax/pagelet/generic.php). D'altro canto, recuperare le richieste `AJAX` porta notevoli vantaggi dal punto di vista informativo poiché tali chiamate restituiscono sempre risposte strutturate (e. g., `JSON` o `XML`).

[55] Applicazione che, dato un indirizzo iniziale, o un insieme di indirizzi iniziali, e delle condizioni iniziali (e. g., numero massimo di indirizzi da percorrere, tipo di pagine da ignorare), scarica tutte le pagine che visita seguendo i *link* tra di esse.

[56] Link: http://en.wikipedia.org/wiki/Robots_exclusion_standard.

[57] `AJAX` è una tecnica per lo scambio di dati in modo asincrono tra *server* e *browser*. Ciò significa che i dati ulteriori sono richiesti al *server* e caricati senza interferire con il comportamento della pagina esistente e già caricata. Questa tecnica consente l'aggiornamento dinamico di una pagina *Web* senza esplicito ricaricamento da parte dell'utente.

Quindi, ogni pagina scovata dal *crawler* con un meccanismo simile a quello descritto verrà poi elaborata dall'applicativo di *scraping*, il quale ha l'obiettivo di trovare ed estrarre le informazioni di interesse dalle pagine `HTML` scaricate.

Tale processo può essere svolto in più modalità (solitamente vengono utilizzate più modalità contemporaneamente): tramite l'utilizzo delle classi `CSS`, imprescindibili per lo sviluppo grafico delle pagine *Web* e quindi sempre presenti, oppure tramite l'ispezione del `DOM`[58] dell'`HTML` della pagina.

Solitamente è conveniente espletare questo processo tramite librerie apposite per il *parsing* delle pagine *Web*, esse infatti sono costruite (e spesso sottoposte a *testing* da anni) per permettere l'iterazione del contenuto della pagina `HTML` come fosse contenuto `JSON`. Tuttavia, in alcuni casi, cioè quando il codice `HTML` del sito *target* non è completamente aderente agli standard, può essere imprescindibile ricorrere all'utilizzo di espressioni regolari che cerchino determinati *pattern* in tutto il codice `HTML` come se fosse una semplice stringa di testo.

Si cita a tal riguardo l'ottima libreria "Beatiful Soup"[59] (`Python`).

4.1 Aggirare le tecniche difensive

A seconda dell'utilizzo che si fa dei dati estratti, i processi di *crawling* e *scraping* possono violare i termini di contratto e di utilizzo del sito *target*. Tali termini sono spesso specificati dal sito *target* tramite l'utilizzo del file `robots.txt`.

Inoltre, i siti che proibiscono esplicitamente il processo di *scraping* (e. g., Facebook) adottano spesso tecniche difensive.

Una delle tecniche difensive più basilari che viene adottata consiste nell'analisi automatica dei *log* del *server* al fine di bloccare i *client* che effettuano una quantità di richieste `HTTP` insolita. Il processo di *scraping* infatti, quando utilizzato, genera quantità notevole di traffico `HTTP` proveniente dall'indirizzo `IP` della macchina che sta eseguendo il relativo processo. Di conseguenza, per gli amministratori dei siti *Web* è abbastanza semplice difendersi semplicemente impostando una soglia di traffico oltre la quale il blocco dell'indirizzo `IP` avvenga automaticamente. Una possibile contromossa consiste nel modificare artificiosamente l'indirizzo `IP` del processo di *scraping* facendolo partire da *client* diversi.

Un'ulteriore tecnica difensiva spesso adottata consiste nel richiedere e verificare che:

- il valore del parametro `User Agent`[60] della richiesta `HTTP` corrisponda a un stringa prefissata
- alla richiesta sia associato un determinato insieme di *Cookie*[61] `HTTP`.

[58] Il `DOM` è l'albero dei tag `HTML` da cui è costituita una pagina *Web*. Si osservi che il `DOM` può essere stato modificato a *run-time* tramite `JavaScript`, il ché implica delle complicazioni nel processo di *parsing* e conseguente estrazioni di dati dalla pagina `HTML`.

[59] "Beautiful Soup": http://www.crummy.com/software/BeautifulSoup.

[60] `User Agent`: stringa indicante il tipo di *client* con cui si accede al *Web*.

[61] I *Cookie* sono stringhe di testo di piccole dimensioni inviate da un *server* ad un *client* (di solito un *browser*) e poi rimandati indietro dal *client* al *server* (senza subire modifiche) ogni volta che il *client* accede alla stessa porzione dello stesso dominio *Web*.

In tal caso la contromossa consiste nell'effettuare uno *spoofing*[62] del sito *target*. A seconda della libreria che si usa per effettuare le richieste `HTTP` (a tal proposito si suggerisce la libreria `Python` "Requests"[63]) tale processo è più o meno semplice: esso consiste semplicemente in una prima e unica chiamata all'*end-point* il cui fine è quello di recuperare la particolare intestazione `HTTP` richiesta (che contiente il valore del parametro `User Agent`) e i *Cookie* richiesti. Una volta ricavati questi valori (o un insieme di tali valori) è possibile utilizzarli per ingannare il sito *target* e fingere che il processo di *scraping* sia effettuato manualmente da degli utenti umani.

Inoltre, la maggior parte delle volte il processo di *login* del sito *target* funge automaticamente da tecnica difensiva di base. In tal caso è infatti necessario possedere un *account* con il quale autenticarsi sul sito *target* prima di avviare il processo di *scraping*. Tuttavia aggirare l'autenticazione semplicemente effettuandola in modo automatico è un facilmente ottenibile tramite varie librerie già esistenti (e. g., "Requests"), anche se ciò renderà tutto il processo tracciabile e associabile a tale *account*.

4.2 Facebook

Facebook adotta una politica estremamente difensiva relativamente allo *scraping* dei suoi dati.

Oltre alle succitate tecniche difensive, Facebook adotta ulteriori tecniche molto avanzate di rilevamento dei *crawler* e dei processi di *scraping*. Ad esempio il loro *software* di difesa genera automaticamente dei codici di verifica nel caso ritenga, dall'analisi dei *log* e dei percorsi di navigazione, che il visitatore non sia un umano bensì un *crawler*. Tale *software* opera anche controlli sull'indirizzo `IP` al fine di verificare che sia incluso in una lista di indirizzi `IP` fissi cui hanno concesso (chiaramente con contratto commerciale) l'esecuzione di processi di *scraping*.

Connettendosi all'`URL` https://facebook.com/robots.txt si ha conferma di tale politica. Infatti, il file `robots.txt` vieta a qualsiasi `User Agent`, che non sia `baiduspider`, `Googlebot`, `msnbot` (e altri omessi), di accedere a qualsiasi percorso (si vedano le ultime due righe dell'estratto riportato di seguito).

```
User-agent: baiduspider
Disallow: /ac.php
Disallow: /ae.php
Disallow: /ajax/
Disallow: /album.php
Disallow: /ap.php
Disallow: /autologin.php
Disallow: /checkpoint/
Disallow: /contact_importer/
Disallow: /feeds/
Disallow: /l.php
Disallow: /o.php
```

[62] Lo spoofing è un tipo di attacco informatico relativo alla falsificazione dell'identità.
[63] Come persistere i *Cookie* durante molte chiamate `HTTP` con "Requests": http://goo.gl/bnVPn.

```
Disallow: /p.php
Disallow: /photo.php
Disallow: /photo_comments.php
Disallow: /photo_search.php
Disallow: /photos.php
Disallow: /sharer/

User-agent: Googlebot
Disallow: /ac.php
Disallow: /ae.php
Disallow: /ajax/
Disallow: /album.php
Disallow: /ap.php
Disallow: /autologin.php
Disallow: /checkpoint/
Disallow: /contact_importer/
Disallow: /feeds/
Disallow: /l.php
Disallow: /o.php
Disallow: /p.php
Disallow: /photo.php
Disallow: /photo_comments.php
Disallow: /photo_search.php
Disallow: /photos.php
Disallow: /sharer/

User-agent: msnbot
Disallow: /ac.php
Disallow: /ae.php
Disallow: /ajax/
Disallow: /album.php
Disallow: /ap.php
Disallow: /autologin.php
Disallow: /checkpoint/
Disallow: /contact_importer/
Disallow: /feeds/
Disallow: /l.php
Disallow: /o.php
Disallow: /p.php
Disallow: /photo.php
Disallow: /photo_comments.php
Disallow: /photo_search.php
Disallow: /photos.php
Disallow: /sharer/

...
```

```
User-agent: *
Disallow: /
```

Il file `robots.txt`, come già specificato, di per sé funge da mero indicatore e non esclude quindi alcuna operazione dal punto di vista tecnico. Perciò è teoricamente possibile, prendendo tutte le dovute precauzioni, effettuare lo *scraping* di Facebook semplicemente non rispettando il succitato *file*. Si consideri tuttavia che tale operazione viola i termini di servizio di Facebook ed è ritenuta illegale e perciò perseguibile nei termini di legge.

È indispensabile notare, comunque, che per effettuale lo *scraping* delle pagine di Facebook è indispensabile considerare adeguatamente l'utilizzo estensivo e avanzato di `AJAX` che tale soggetto effettua per il proprio sito *Web*. Ad esempio, quando si contatta la pagina *Web* relativa agli amici di un utente (sempre nel caso in cui le sue impostazioni di *privacy* rendano visibile tale pagina), cioè `https://facebook.com/<username>/friends`, Facebook restituisce una pagina popolata con massimo 400 amici. Sarà perciò necessario simulare più volte lo *scrolling* (azione che, tramite chiamata asincrona, recupera altri dati dai *server*) della pagina *Web* al fine di ottenere altri blocchi di 400 amici.

Si mostra di seguito un tipico flusso di *crawling* operabile al fine di estrarre la lista completa degli amici di un utente Facebook (Catanese, Meo, and Ferrara 2011).

N.	Azione	Protocollo	Metodo	URI
1	Accedere a Facebook	`HTTP`	`GET`	facebook.com
2	Autenticazione	`HTTPS`	`POST`	login.facebook.com
3	Accesso agli amici	`HTTPS`	`GET`	facebook.com/home.php
4	Accesso ulteriore	`HTTPS/AJAX`	`GET`	facebook.com/ajax/pagelet/generic.php

Tabella 1: Passi per il *crawling* della lista di amici di un utente Facebook

Si osservi che sarà necessario ripetere il passo `4` un numero di volte sufficiente a caricare totalmente la lista degli amici Facebook dell'utente in esame.

A questo punto sarà possibile ottenere la pagina `HTML` completa e estrarre (i. e., *scraping*) varie informazioni riguardo agli amici: nominativi, `username` e quindi i *link* ai rispettivi profili Facebook. Nel caso si voglia automatizzare questo processo per tutti gli utenti, il *crawler* dovrà semplicemente inserire i nominativi estratti in ogni pagina in una coda da cui attingere per i nuovi profili da analizzare.

Si fa notare, inoltre che esistono altri *end-point* per ottenere informazioni di questo tipo. Ad esempio una sorgente enorme di informazioni sugli utenti di Facebook è la sua *directory*[64]. Essa permette di sfogliare per nome e alfabeticamente la lista degli

[64] *Facebook People Directory*: https://www.facebook.com/directory/people

oltre 1 miliardo di utenti. Allo stesso modo esiste anche una *directory* relativa alle pagine[65] e ai posti[66] (i. e., *locations* o *places*).

Purtroppo questa sorgente di informazione, dopo essere stato scoperta e utilizzata estensivamente, è, al momento, sottoposta anch'essa al monitoraggio del *software* anti estrazione con regole molto restrittive: dopo qualche centinaio di chiamate `HTTP` il proprio `IP` viene automaticamente bloccato da Facebook.

Un approccio simile a quello appena esposto consiste invece nello sfruttare la *sitemap*[67] di Facebook, bloccata di default. Per avervi accesso è necessario scrivere all'email <sitemaps@lists.facebook.com>. Non è dato sapere nessun altro particolare circa questo argomento.

Oltre che per l'accesso alla *sitemap*, si noti che è possibile richiedere a Facebook il permesso di collezionare automaticamente i suoi dati. A tale scopo è necessario sottostare ai relativi termini di servizio[68] e inoltrare la richiesta contattando Facebook al seguente indirizzo: https://www.facebook.com/apps/site_scraping_tos.php.

4.2.1 Dataset

La conferma concreta che il processo di *scraping* di Facebook è possibile, seppur molto ostico, deriva direttamente dal fatto che, in letteratura, esistono prove tangibili del fatto che tale processo è stato effettuato e portato a termine con successo in più casi. Si veda, a tal riguardo, Catanese, Meo, and Ferrara (2011), Traud et al. (2008) e Traud, Mucha, and Porter (2011).

Inoltre, Traud, Mucha, and Porter (2011) hanno reso disponibile al pubblico i *dataset* utilizzati per le relative ricerche scientifiche: un *dataset* composto dal grafo sociale degli studenti di oltre 100 università USA. Si osservi che tale *dataset* corrisponde a una istantanea di Facebook nel settembre 2005.

È importante notare come il *Facebook Data Team* abbia sempre combattuto il rilascio di *dataset* di tale tipo, ottenendo sempre la loro rimozione dal *Web*. Tuttavia, a causa della natura del *Web* è tutt'oggi possibile ricavare due versioni di tale *dataset*:

- una sua versione completa, chiamata `facebook100v1`, reperibile all'indirizzo: http://www.monova.org/torrent/4266013/facebook100_zip.html
- una versione senza gli `id` degli utenti, chiamata `facebook100v2`, reperibile all'indirizzo: http://archive.org/details/oxford-2005-facebook-matrix.

Per maggiori informazioni riguardanti questo *dataset* è possibile consultare i seguenti indirizzi: 1[69], 2[70], 3[71].

[65] *Facebook Pages Directory*: https://www.facebook.com/directory/pages
[66] *Facebook Places Directory*: https://www.facebook.com/directory/pages
[67] *Facebook Sitemap*: http://www.facebook.com/sitemap.php
[68] *Site Scraping TOS*: https://www.facebook.com/apps/site_scraping_tos_terms.php.
[69] http://masonporter.blogspot.it/2011/02/facebook100-data-set.html
[70] http://sociograph.blogspot.it/2011/02/visualizing-large-facebook-friendship.html
[71] http://sociograph.blogspot.it/2011/03/facebook100-data-and-parser-for-it.html

A Codice R

A.1 Interrogare le Graph API

In questa sezione si riporta il codice `R` utilizzato per implementare gli esempi relativi all'estrazione di informazioni da Facebook.
Di seguito si mostrano le librerie richieste e come caricarle in `R`.

```
library(RCurl)
library(rjson)
```

Mentre, la funzione per l'interazione (in lettura) con le *Graph API* di Facebook è la seguente. I commenti riportano

```
#' Chiama le Graph API di Facebook.
#'
#' Questa funzione permette di effettuare una chiamata alle API di Facebook.
#' Permette di specificare tutti i possibili componenti: percorso, opzioni e access token.
#'
#' @param   path        Percorso della chiamata [default 'me'].
#' @param   token       Access token per la chiamata.
#' @param   opts        Lista nominata dei parametri della chiamata.
#'
#' @return  Una lista nominata contenente i campi e i valori restituiti dalle Graph API.
facebook <- function(path = "me", token, opts) {
    if (!missing(opts)) {
        if (!missing(token)) {
            opts <- append(opts, list(access_token = token))
        }
        opts <- sapply(names(opts), function(i) paste(curlEscape(opts[[i]]), collapse = ","))
        opts <- paste(names(opts), "=", opts, sep = "", collapse = "&")
        opts <- sprintf("?%s", opts)
    } else {
        opts <- ifelse(missing(token), "", paste("?access_token=", token, sep = ""))
    }
    url <- sprintf("https://graph.facebook.com/%s%s", path, opts)
    res <- getURL(url)
    return(fromJSON(res))
}
```

A.2 Autenticazione application-only in Twitter API v1.1

In questa sezione si riporta il codice `R` utilizzato per effettuare un'autenticazione di tipo *application-only* nelle Twitter `API` (versione 1.1).

Innanzitutto, si mostrano le dipendenze.

```
library(RCurl)
library(ROAuth)
library(twitteR)
```

Di seguito invece vengono riportare le funzioni utili ad effettuare l'autenticazione *application-only* (i. e., *OAuth 2.0*), salvare le credenziali ed eventualmente ricaricarle per utilizzi successivi (si ricorda che la finestra temporale che Twitter mette a disposizione è di 15 minuti).

```r
#' Loads stored twitter credentials.
#'
#' The file from which the credentials are loaded must be in the working directory.
#' Furthermore, it have to be called 'twitter_credentials' (without extension).
#'
#' @return  A boolean indicating if the loaded credentials are still valid.
load_twitter_creds <- function() {
    load("twitter_credentials")
    return(registerTwitterOAuth(creds))
}

#' Performs application-only authentication.
#'
#' The application is identified by the consumer key and secret.
#' Credentials are stored in a file named 'twitter_credentials' inside the working directory.
#'
#' @param   key     The consumer key of the Twitter application.
#' @param   secret  The consumer secret of the Twitter application.
#' @param   store   Whether to store the certificate and the credentials [default = TRUE].
#' @return  A boolean indicating if application-only authorization is successfull or not.
twitter_app_only_auth <- function(key, secret, store = TRUE) {
    req_url <- "https://api.twitter.com/oauth/request_token"
    acc_token_url <- "https://api.twitter.com/oauth/access_token"
    auth_url <- "https://api.twitter.com/oauth/authorize"
    creds <- OAuthFactory$new(consumerKey = key, consumerSecret = secret, authURL = auth_url,
        requestURL = req_url, accessURL = acc_token_url)
    if (!file.exists("cacert.perm")) {
        download.file(url = "http://curl.haxx.se/ca/cacert.pem", destfile = "cacert.perm")
    }
    certificate <- system.file("CurlSSL", "cacert.pem", package = "RCurl")
    creds$handshake(cainfo = certificate)
    res <- registerTwitterOAuth(creds)
    if (store) {
        save(list = "creds", file = "twitter_credentials")
    } else {
        unlink("cacert.perm", force = TRUE)
    }
    return(res)
}
```

La funzione `twitter_app_only_auth` effettua le seguenti operazioni:

1. crea un oggetto *OAuth* tramite una classe *factory* fornita dal pacchetto `ROAuth` (Gentry and Lang 2013)
2. scarica il certificato per la connessione `SSL` nel caso esso non sia già stato scaricato, tramite funzionalità di base di `R` (R Core Team 2013)
3. effettua il processo di *hand-shaking* con il *server* (sempre tramite il pacchetto `ROAuth`)
4. registra le credenziali tramite il metodo `registerTwitterOAuth` del pacchetto `twitteR` (Gentry 2013)
5. salva le credenziali (qualora richiesto) per eventuali utilizzi successivi.

A.3 Sessione R

Informazioni sulla sessione `R` utilizzata.

```
sessionInfo()
```

```
# R version 3.0.1 (2013-05-16)
# Platform: x86_64-pc-linux-gnu (64-bit)
#
# locale:
#  [1] LC_CTYPE=it_IT.UTF-8       LC_NUMERIC=C               LC_TIME=it_IT.UTF-8
#  [4] LC_COLLATE=it_IT.UTF-8     LC_MONETARY=it_IT.UTF-8    LC_MESSAGES=it_IT.UTF-8
#  [7] LC_PAPER=C                 LC_NAME=C                  LC_ADDRESS=C
# [10] LC_TELEPHONE=C             LC_MEASUREMENT=it_IT.UTF-8 LC_IDENTIFICATION=C
#
# attached base packages:
# [1] methods   stats     graphics  grDevices utils     datasets  base
#
# other attached packages:
# [1] codetools_0.2-8 stringr_0.6.2   twitteR_1.1.7   ROAuth_0.9.3    RCurl_1.95-4.1
# [6] bitops_1.0-5    rjson_0.2.12    digest_0.6.3    knitr_1.2.13
#
# loaded via a namespace (and not attached):
# [1] evaluate_0.4.4 formatR_0.8    tools_3.0.1
```

B Esempi di estrazione dati

B.1 Creare una mappa geo-taggata degli amici Facebook

Si presenta un esempio articolato di estrazione dei dati da Facebook.

Nello specifico il codice `R` presentato di seguito mostra come, ottenuta la lista di amici (in questo esempio limitata a 50 unità) di un determinato utente Facebook (variabile `friend`), sia possibile ottenere, per ognuno di essi, la lista dei post geo-taggati (variabile `friend_locs`). Si osservi che per tale operazione l'utente deve fornire i permessi "friends_location".

Infine, è possibile, tramite la libreria `ggplot2` (Wickham 2009), visualizzare l'insieme di tutti i punti (latitudine, longitudine) ottenuti su una mappa; così da avere una visione d'insieme dei posti geografici frequentati della propria cerchia di amici.

In primis, si riportano due funzioni `R` utili a lavorare (e. g., filtrare) le liste. La documentazione di tali funzioni riporta maggiori dettagli.

```
#' Flattens a nested array (the nesting can be to any depth).
#'
#' @param   ls          The list to be flattened.
#' @param   shallow     Whether or not flatten recursively [default 'FALSE'].
#' @return  A named vector.
flatten <- function(ls, shallow = TRUE) {
    return(unlist(ls, recursive = shallow))
}

#' A convenient version of what is perhaps the most common use-case for map:
#' extracting a list of property values.
#'
#' @param   ls          The subject list.
#' @param   field_name  The name of the field to be extracted (can also be a list of names).
#' @return  A named vector where each name is equal or contained to the input field name/s.
pluck <- function(ls, field_name) {
    ls <- flatten(ls)
    if (is.list(field_name)) {
```

```
        field_name <- paste(field_name, collapse = "|", sep = "")
    }
    res <- ls[grepl(field_name, names(ls))]
    return(res)
}
```

Di seguito, invece, si presenta il codice necessario ad estrarre i post geo-taggati appartenenti a 50 amici dell'utente "leodido". Tale porzione di codice crea quindi una lista (un elemento per ogni amico) di tabelle (i. e., `data.frame`) contenenti la latitudine e longitudine di ogni post geo-taggato recuperato.

```
# token <- '...'
friends <- facebook("leodido/friends", token = token, opts = list(limit = 50))$data
friend_ids <- pluck(friends, "id")
locations <- lapply(friend_ids, function(friend_id) {
    locs_path <- paste(friend_id, "/locations", sep = "")
    friend_locs <- facebook(locs_path, token = token)
    if (length(friend_locs) > 0) {
        friend_locs <- friend_locs$data
        lon <- as.numeric(pluck(friend_locs, "place.location.longitude"))
        lat <- as.numeric(pluck(friend_locs, "place.location.latitude"))
        return(data.frame(x = lon, y = lat))
    } else {
        return(NA)
    }
})
locations <- unname(locations[!is.na(locations)])
```

Quindi si visualizza, tramite la libreria `ggplot2` (Wickham 2009) (che è necessario caricare per questo specifico esempio), la lista contenente le informazioni geografiche relative ai post degli amici.

Si osservi, tuttavia, che la politica di Facebook relativamente alla privacy fa sì che *Graph API* restituiscano un massimo di 25 post geo-taggati per ogni utente.

Di seguito il codice dedito alla creazione della mappa delle *locations*.

```
suppressPackageStartupMessages(library(ggplot2))
map_plot <- ggplot(legend = FALSE)
map_poly <- geom_polygon(data = map_data("world"), aes(x = long, y = lat, group = group))
map_plot <- map_plot + map_poly
for (i in seq.int(length(locations))) {
    places <- locations[[i]]
    map_plot <- map_plot + geom_point(data = places, aes(x, y), colour = "orange")
}
print(map_plot)
```

Ecco quindi l'output di questo esempio:

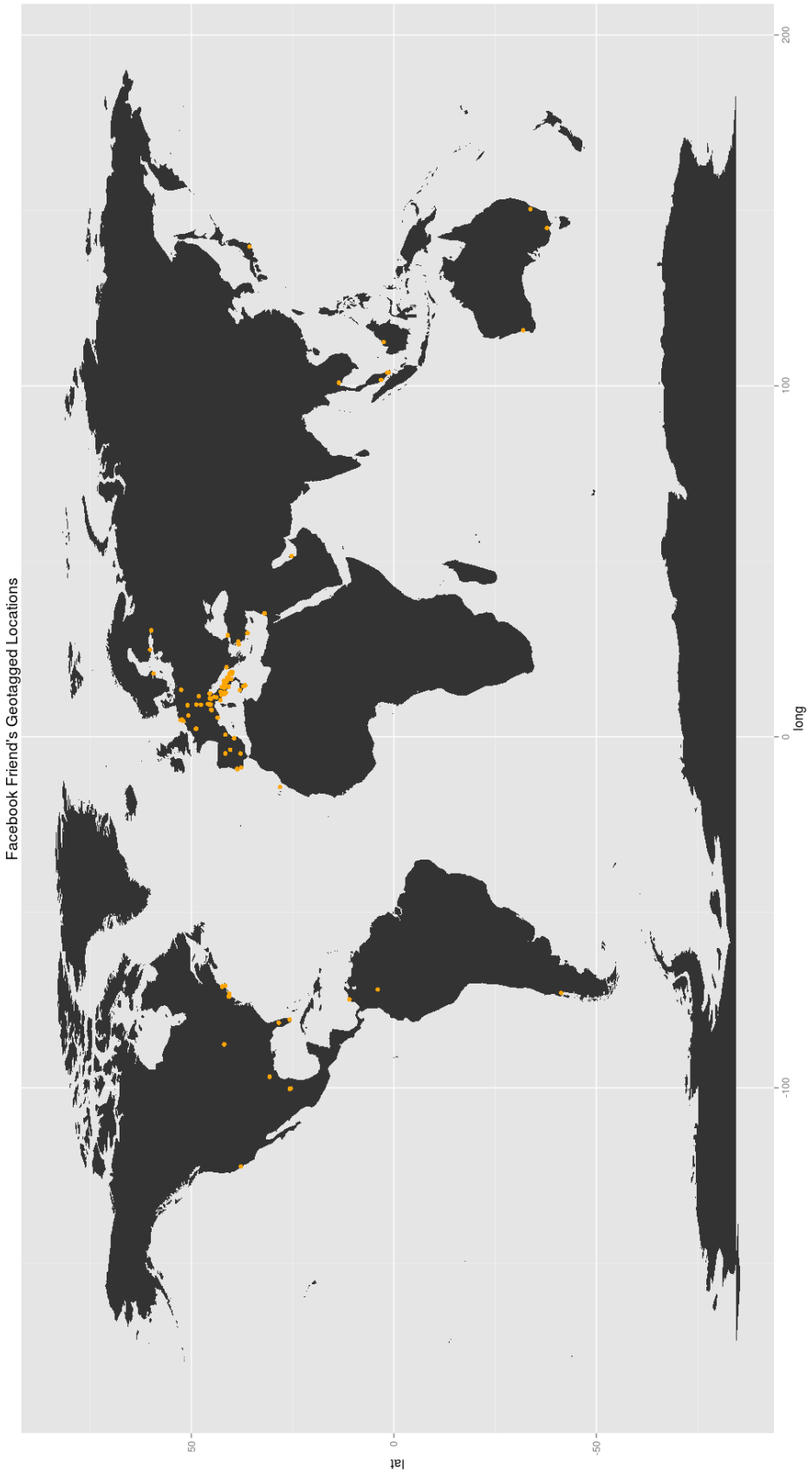

Figura 10: Mappa che visualizza tutti i post geo-taggati degli amici dell'utente leodido

B.2 Ricercare in Twitter con autenticazione per conto dell'utente

Questo esempio ha due obiettivi:

1. mostrare l'autenticazione *application-user* (i. e., per conto di un utente)
2. mostrare come costruire ed effettuare una richiesta `HTTP` verso l'*end-point* di ricerca.

In primis è necessario caricare i seguenti pacchetti `R`: `stringr`, `rjson`, `RCurl` e `digest`.

```
library(digest)
library(stringr)
library(rjson)
library(RCurl)
```

Quindi:

1. si verifica che il certificato utile all'autenticazione sia presente della corrente cartella di lavoro, altrimenti lo si scarica
2. si impostano le credenziali Twitter: *consumer key*, *consumer secret*, *access token*, *access token secret*[72]
3. si registra la posizione del certificato e alcune opzioni utili alla connessione `SSL` con `curl`.

```
if (!file.exists("cacert.perm")) {
    download.file(url = "http://curl.haxx.se/ca/cacert.pem", destfile = "cacert.perm")
}

tw_creds <- list()
tw_creds[["oauth_key"]] <- tw_token
tw_creds[["oauth_secret"]] <- tw_token_secret
tw_creds[["consumer_secret"]] <- tw_app_secret
tw_creds[["consumer_key"]] <- tw_app_key

curl <- getCurlHandle()
options(RCurlOptions = list(capath = system.file("CurlSSL", "cacert.pem", package = "RCurl"),
    ssl.verifypeer = FALSE))
invisible(curlSetOpt(.opts = list(proxy = "proxyserver:port"), curl = curl))
```

Di seguito invece la funzione dedita all'autenticazione per conto dell'utente (identificato dall'*access token* fornito) e alla ricerca tramite le *Twitter Search API*.

```
twitter_search_api <- function(term, count = 5, page_info = "") {
    term <- curlPercentEncode(term)
    query <- paste0("&q=", term)

    url_params <- paste0(paste0("count=", count), "&lang=en", page_info)
    url <- "https://api.twitter.com/1.1/search/tweets.json"
    uri_extra <- paste("q=", term, "&", url_params, sep = "")

    baseString <- paste(curlPercentEncode(c("GET", url, url_params)), collapse = "&")
```

[72] Si veda la sezione 3.1.3 per maggiori informazioni circa la loro creazione.

```
    nonce <- paste(letters[runif(34, 1, 27)], collapse = "")
    timestamp <- as.integer(Sys.time())
    ckey <- tw_creds$consumer_key
    sign_method <- "HMAC-SHA1"
    token <- tw_creds$oauth_key

    param_str <- paste0("&oauth_consumer_key=", ckey)
    param_str <- paste0(param_str, "&oauth_nonce=", nonce)
    param_str <- paste0(param_str, "&oauth_signature_method=", sign_method)
    param_str <- paste0(param_str, "&oauth_timestamp=", timestamp)
    param_str <- paste0(param_str, "&oauth_token=", token)
    param_str <- paste0(param_str, "&oauth_version=1.0")
    param_str <- paste0(param_str, query)

    signature_base_str <- paste0(baseString, curlPercentEncode(param_str))
    enc_consumer_secret <- curlPercentEncode(tw_creds$consumer_secret)
    enc_oauth_secret <- curlPercentEncode(tw_creds$oauth_secret)
    signing_key <- paste(enc_consumer_secret, "&", enc_oauth_secret, sep = "")
    signature <- hmac(signing_key, signature_base_str, algo = "sha1", serialize = FALSE,
        raw = TRUE)
    signature <- curlPercentEncode(base64(signature))

    auth_header <- paste0("Authorization: OAuth ", "oauth_consumer_key=\"", ckey, "\", ")

    auth_header <- paste0(auth_header, "oauth_nonce=\"", nonce, "\", ")
    auth_header <- paste0(auth_header, "oauth_signature=\"", signature, "\", ")
    auth_header <- paste0(auth_header, "oauth_signature_method=\"", sign_method, "\", ")
    auth_header <- paste0(auth_header, "oauth_timestamp=\"", timestamp, "\", ")
    auth_header <- paste0(auth_header, "oauth_token=\"", token, "\", ")
    auth_header <- paste0(auth_header, "oauth_version=\"1.0\"")

    .opts <- list(header = TRUE, httpauth = TRUE, verbose = TRUE, ssl.verifypeer = FALSE)
    uri <- paste0(url, "?", uri_extra)

    result <- getURI(uri, .opts = .opts, httpheader = auth_header)
    result <- fromJSON(str_match_all(result, "\\r\\n\\r\\n(\\{.+)")[[1]][1, 2])
    return(result)
}
```

La funzione appena presentata effettua una richiesta alle *Twitter Search API* effettuando l'autenticazione per conto dell'utente (i. e., rappresentato dall'*access token*). I passi che essa compie sono i seguenti:

1. operazioni preliminari sui parametri
2. creazione dell'indirizzo `HTTP` di base da chiamare in `GET`
3. creazione del *nonce* e del *timestamp*
4. creazione della stringa dei parametri (comprensio il parametro `q` contenente la *query* di ricerca)
5. firma della richiesta tramite il metodo `hmac` del pacchetto `digest` (Dirk Eddelbuettel 2013), così come richiesto dalla specifica *OAuth 1.0a*
6. creazione dell'intestazione di autorizzazione per la richiesta `HTTP`
7. esecuzione della richiesta `HTTP GET`

8. *post-processing* dei risultati tramite la funzione `str_match_all` del pacchetto `stringr` (Wickham 2012)

9. conversione dei *record* ottenuti in un oggetto `JSON` e restituzione di tale oggetto.

Una volta autenticati possiamo quindi eseguire una qualsiasi ricerca. Ad esempio potremmo voler individuare i primi 10 *tweet* rilevanti per la *query* "user2013".

```
res <- twitter_search_api("user2013", count = 10)
```

Quindi si stampano sia i metadati (e. g., tempo d'esecuzione) che i risultati della ricerca nelle modalità che più si preferisce.

```
print(res$search_metadata)
```

```
# $completed_in
# [1] 0.042
#
# $max_id
# [1] 3.577e+17
#
# $max_id_str
# [1] "357684389977022464"
#
# $next_results
# [1] "?max_id=357224135987953663&q=user2013&lang=en&count=10&include_entities=1"
#
# $query
# [1] "user2013"
#
# $refresh_url
# [1] "?since_id=357684389977022464&q=user2013&lang=en&include_entities=1"
#
# $count
# [1] 10
#
# $since_id
# [1] 0
#
# $since_id_str
# [1] "0"
```

```
tweets <- res$statuses
for (i in seq.int(length(tweets))) {
    tweet <- tweets[[i]]
    date <- tweet$created_at
    username <- tweet$user$screen_name
    text <- paste(substring(tweet$text, 0, 80), "...")
    cat(sprintf("(%d)\n[%s]\n<@%s> tweeted:\n%s.\n\n", i, date, username, text))
}
```

```
# (1)
# [Thu Jul 18 02:12:58 +0000 2013]
# <@fellgernon> tweeted:
# RT @revodavid: Speedup #rstats code by inlining C++ -- @revojoe's #useR2013 expe ....
#
# (2)
# [Wed Jul 17 22:44:28 +0000 2013]
# <@eoinbrazil> tweeted:
# RT @revodavid: Speedup #rstats code by inlining C++ -- @revojoe's #useR2013 expe ....
```

```
#
# (3)
# [Wed Jul 17 21:56:25 +0000 2013]
# <@divyakkm> tweeted:
# RT @revodavid: Speedup #rstats code by inlining C++ -- @revojoe's #useR2013 expe ....
#
# (4)
# [Wed Jul 17 21:51:45 +0000 2013]
# <@eddelbuettel> tweeted:
# RT @revodavid: Speedup #rstats code by inlining C++ -- @revojoe's #useR2013 expe ....
#
# (5)
# [Wed Jul 17 21:46:40 +0000 2013]
# <@revodavid> tweeted:
# Speedup #rstats code by inlining C++ -- @revojoe's #useR2013 experiences with Rc ....
#
# (6)
# [Wed Jul 17 17:01:02 +0000 2013]
# <@CornelissenJo> tweeted:
# RT @statslabdublin: http://t.co/EYilMw5Dpu : looks quite interesting useful teac ....
#
# (7)
# [Wed Jul 17 15:45:24 +0000 2013]
# <@ConcejeroPedro> tweeted:
# Testing two treatments against a control (3 log regression) using trimatch, from ....
#
# (8)
# [Wed Jul 17 12:00:43 +0000 2013]
# <@southmapr> tweeted:
# @useR_2013 Racy advertising for #rstats executable, targeting #user2013 attendee ....
#
# (9)
# [Tue Jul 16 23:46:57 +0000 2013]
# <@LouBajuk> tweeted:
# Presentation by Michael Sannella @TIBCOSpotfire on Memory Management in @TIBCO E ....
#
# (10)
# [Tue Jul 16 19:44:05 +0000 2013]
# <@adolfoalvarez> tweeted:
# RT @southmapr: @robjhyndman Thanks for including my talk in your reflections on  ....
```

Si osservi che è possibile recuperare ulteriori risultati dalla pagina successiva tramite il parametro `page_info`, e. g. `&max_id=<xxx>`, dove `<xxx>` è l'`id` presente nell'output, specificatamente nel campo `res$search_metadata$next_results`.

Con questo esempio si è voluto evidenziare la difficoltà (e il maggior grado di sicurezza) imposto dal protocollo *OAuth 1.0a*, a differenza del protcollo `OAuth 2.0`. Per tale motivo non si è utilizzato alcun libreria di supporto alla creazione dell'autenticazione *OAuth 1.0a* per conto dell'utente.

Allo scopo di semplificare l'esempio in tale direzione si consiglia la riscrittura della funzione `twitter_search_api` utilizzando i metodi *OAuth* del pacchetto `httr` (Wickham 2013).

C Addendum

C.1 Limiti della Twitter API

I seguenti limiti sono da intendersi per ogni finestra di 15 minuti.

Famiglia	Risorsa specifica	RPAUA	RPAOA
account	`account/settings`	15	
account	`account/verify_credentials`	15	
application	`application/rate_limit_status`	180	180
blocks	`blocks/ids`	15	
blocks	`blocks/list`	15	
direct_messages	`direct_messages`	15	
direct_messages	`direct_messages/sent`	15	
direct_messages	`direct_messages/show`	15	
favorites	`favorites/list`	15	15
followers	`followers/ids`	15	15
followers	`followers/list`	15	30
friends	`friends/ids`	15	15
friends	`friends/list`	15	30
friendships	`friendships/incoming`	15	
friendships	`friendships/lookup`	15	
friendships	`friendships/no_retweets/ids`	15	
friendships	`friendships/outgoing`	15	
friendships	`friendships/show`	180	15
geo	`geo/id/:place_id`	15	
geo	`geo/reverse_geocode`	15	
geo	`geo/search`	15	
geo	`geo/similar_places`	15	
help	`help/configuration`	15	15
help	`help/languages`	15	15
help	`help/privacy`	15	15
help	`help/tos`	15	15
lists	`lists`	15	
lists	`lists/list`	15	15
lists	`lists/member`	180	15
lists	`lists/members/sho`	15	15
lists	`lists/membership`	15	15

lists	`lists/ownership`	15	15
lists	`lists/sho`	15	15
lists	`lists/statuse`	180	180
lists	`lists/subscriber`	180	15
lists	`lists/subscribers/sho`	15	15
lists	`lists/subscription`	15	15
saved_searches	`saved_searches/lis`	15	
saved_searches	`saved_searches/show/:i`	15	
search	`search/tweet`	180	450
statuses	`statuses/home_timeline`	15	
statuses	`statuses/mentions_timeline`	15	
statuses	`statuses/oembed`	180	180
statuses	`statuses/retweeters/ids`	15	60
statuses	`statuses/retweets/:id`	15	60
statuses	`statuses/retweets_of_me`	15	
statuses	`statuses/show/:id`	180	180
statuses	`statuses/user_timeline`	180	300
trends	`trends/available`	15	15
trends	`trends/closest`	15	15
trends	`trends/place`	15	15
users	`users/contributees`	15	
users	`users/contributors`	15	
users	`users/lookup`	180	60
users	`users/profile_banner`	180	
users	`users/search`	180	
users	`users/show`	180	180
users	`users/suggestions`	15	15
users	`users/suggestions/:slug`	15	15
users	`users/suggestions/:slug/members`	15	15
users	`users/suggestions/:slug`	15	15

Tabella 2: Massimo numero di richieste per risorsa Twitter

Bibliografia

Barbera, Pablo. 2013. *streamR: Access to Twitter Streaming API via R*. http://CRAN.R-project.org/package=streamR.

Catanese, S. A., P. De Meo, and Emilio Ferrara. 2011. "Crawling facebook for social network analysis purposes." *Proceedings of the International Conference on Web Intelligence, Mining and Semantics*: 1. doi:10.1145/1988688.1988749.

Couture-Beil, Alex. 2013. *rjson: JSON for R*. http://CRAN.R-project.org/package=rjson.

Crockford, Douglas. 2006. "The application/json Media Type for JavaScript Object Notation (JSON)." http://tools.ietf.org/tools/rfcmarkup/rfcmarkup.cgi?rfc=4627.

Dirk Eddelbuettel, Antoine Lucas, Jarek Tuszynski, Henrik Bengtsson, Simon Urbanek, Mario Frasca, Bryan Lewis, Murray Stokely, Hannes Muehleisen, Duncan Murdoch. 2013. *digest: Create cryptographic hash digests of R objects*. http://CRAN.R-project.org/package=digest.

Fielding, Roy Thomas. 2000. "Architectural styles and the design of network-based software architectures."

Gentry, Jeff. 2013. *twitteR: R based Twitter client*. http://CRAN.R-project.org/package=twitteR.

Gentry, Jeff, and Duncan Temple Lang. 2013. *ROAuth: R interface for OAuth*. http://CRAN.R-project.org/package=ROAuth.

Hammer-Lahav, Eran. 2010. "The OAuth 1.0 Protocol." http://tools.ietf.org/html/rfc5849.

Hardt, Dick. 2012. "The OAuth 2.0 Authorization Framework." http://tools.ietf.org/html/rfc6749.

Lang, Duncan Temple. 2013. *RCurl: General network (HTTP/FTP/...) client interface for R*. http://CRAN.R-project.org/package=RCurl.

R Core Team. 2013. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. http://www.R-project.org/.

Traud, Amanda L., Eric D. Kelsic, Peter J. Mucha, and Mason A. Porter. 2008. "Comparing Community Structure to Characteristics in Online Collegiate Social Networks." *SIAM review* 53 (3): 17. doi:10.1137/080734315.

Traud, Amanda L., Peter J. Mucha, and Mason A. Porter. 2011. "Social Structure of Facebook Networks." *CoRR*: 82.

Wickham, Hadley. 2009. *ggplot2: elegant graphics for data analysis*. Springer New York. http://had.co.nz/ggplot2/book.

———. 2012. *stringr: Make it easier to work with strings*. http://cran.r-project.org/web/packages/stringr/index.html.

———. 2013. *httr: Tools for working with URLs and HTTP*. https://github.com/hadley/httr.