Note sullo studio del backend

I veri dettagli da discutere sono pochi, e sono commentati all'interno del codice. Ho commentato i dettagli solo la prima volta che compaiono nel codice, considerandolo in ordine alfabetico (ad esempio, se lo stesso dettaglio comparisse in authenticationController.js e in spotifyController.js, il commento che lo segnala sarebbe in authenticationController.js). Qui invece segnalerò alcuni concetti chiave che sono imprescindibili per la comprensione del funzionamento del backend, in particolare il concetto di **Promise**, del quale si fa un uso abbastanza diffuso, il flusso di **oAuth**, implementato in diverse parti del codice, il concetto di **Modello** e la struttura del codice, ovvero **la relazione tra Router, Modelli e Controller.** Queste mi sembrano le uniche informazioni necessarie alla comprensione, ma se ne sorgono altre segnalatemele e vedrò di aggiungerle in questo file.

Promises

Il problema della programmazione asincrona è che operazioni più lente, come accessi a memoria o richieste alla rete, richiedono un tempo enorme per arrivare a compimento rispetto ad operazioni più semplici effettuate su RAM: per questa ragione, in un ambiente asincrono come NodeJS, per effettuare una operazione quando si è certi che la richiesta abbia terminato di fare ciò che doveva (leggere in memoria, è arrivata la risposta dalla rete...) esistono due opzioni, le callback e le promises.

Le callback, estremamente diffuse nel nostro backend, sono delle funzioni che vengono eseguite quando l'operazione primaria è arrivata a compimento. Sono in genere indicate nei parametri stessi della funzione (sono, ad esempio, l'ultimo parametro passato ad una qualsiasi request.get che trovate nel codice).

Le Promises invece sono un tipo di funzione, e la logica che le governa è questa:

- Voglio che venga fatta una operazione, che però può richiedere dei tempi di esecuzione molto lunghi;
- Quando quella funzione è fatta, ne analizzo il risultato. Se è quello corretto, passo questo risultato alla funzione resolve(), che a breve vedremo. Se il risultato è sbagliato, passo alla funzione reject() qualcosa che mi interessa riguardo all'errore avvenuto.

Ora, se io ho usato una Promise per implementare una certa logica, quello che posso fare per assicurarmi che una qualche operazione venga svolta SICURAMENTE dopo che la funzione ha completato la sua esecuzione, è scrivere codici in questa forma (supponiamo che la funzione realizzata come promise si chiami promiseFunc):

promiseFunc.

then((result) => cosa succede se la funzione ha avuto successo)

catch((error) =>cosa succede se la funzione non ha avuto successo)

I due parametri catturati dalle funzioni then e catch sono, rispettivamente, i valori passati alle funzioni resolve e reject all'interno della definizione della Promise.

Su internet è possibile trovare, ovviamente, guide più dettagliate e tecniche (e formalmente più corrette) di quanto visto qua, ma credo che per lo studio del backend questo possa bastare.

oAuth

Consiglio la pagina Wikipedia, sinceramente. Anche Youtube è pieno zeppo di video che spiegano il funzionamento di oAuth.

La ragione essenziale dell'esistenza di oAuth è questa: è un protocollo che permette di accedere a dati sensibili di un utente (mail, password, account Facebook, Google, Twitter...), previa sua autorizzazione, senza dover mai maneggiare le credenziali stesse dell'utente. Ha successo principalmente per due ragioni:

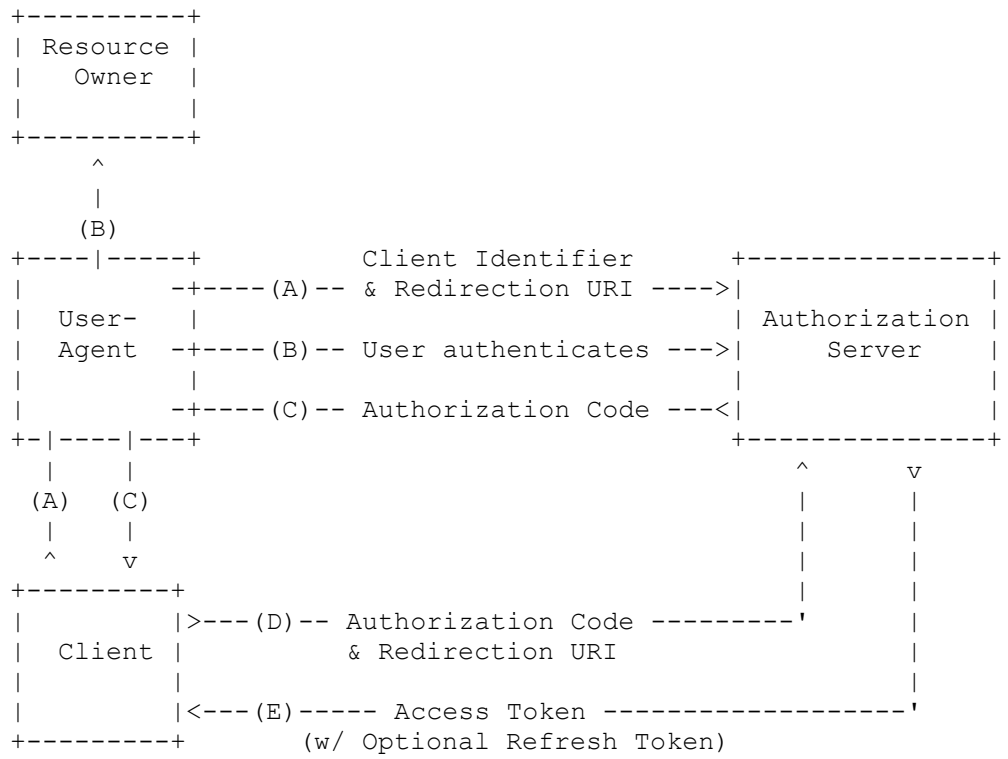
- L'utente si "fida" di dare l'autorizzazione ad una app perché, quando va a dare il permesso, vede chiaramente il tipo di azioni che sta permettendo di fare alla web app che vuole usare, ad esempio, il suo account Twitter. Non è un lasciapassare totale che dice "Questo è il mio account Twitter, puoi farne ciò che vuoi". Il tipo di azioni che l'utente permette di svolgere alla app che lo richiede prende il nome di **scope** della richiesta;
- Dal punto di vista di chi realizza l'app, è una comodità: non deve infatti salvare le credenziali di ciascuno dei suoi utenti e non deve preoccuparsi di proteggerle, nasconderle, criptarle e quanto altro. Inoltre, è sicuro che non farà (in teoria) nulla di illegale o non voluto dall'utente, perché è limitato dallo **scope** che ha chiesto all'utente stesso.

oAuth è descritto da un flusso (**oAuth flow**), che viene implementato da tutte le app che vogliono offrire questa possibilità. Al flusso partecipano diversi attori:

- Resource Owner è l'utente;
- User Agent è, nel nostro caso ma anche abbastanza generalmente, il browser;

- Authorization Server è il server che si occupa di controllare che l'utente dia il permesso per un certo scope e di fornire il token adatto per poter maneggiare le risorse incluse in quello scope;
- Client è, nel nostro caso, il nostro backend server. Qui prende il nome di client perché, essendo lui a mandare una richiesta per accedere alle risorse private dell'utente all'authorization server, dal punto di vista di quest'ultimo è effettivamente un client.

La figura qua sotto è presa dall'RFC che definisce oAuth 2.0 (la differenza tra 2.0 e 1.0 è che è stato aggiunto un intermediatore tra client e server per ragioni di sicurezza e che i token possono avere expiration date maggiori, abbastanza irrilevante per noi).



Quello che sta accadendo nella figura è spiegato, sempre nell'RFC, così:

A) The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

(B) The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

(C) Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an authorization code and any local state provided by the client earlier.

(D) The client requests an access token from the authorization server's token endpoint by including the authorization code received in the previous step. When making the request, the client authenticates with the authorization server. The client includes the redirection URI used to obtain the authorization code for verification.

(E) The authorization server authenticates the client, validates the authorization code, and ensures that the redirection URI received matches the URI used to redirect the client in step (C). If valid, the authorization server responds back with an access token and, optionally, a refresh token.

Una volta che si è ottenuto il token descritto nei passi D ed E, si possono effettuare richieste ad un Resource Server (può essere lo stesso server che fa da authorization server, può essere un altro, assolutamente irrilevante) aggiungendo ad ogni richiesta il token: funziona così come una sorta di documento di identità che dice:“Sono l’applicazione X per conto dell’utente Y, e come definito da questo token ho accesso a questi SCOPE. Vorrei ottenere questa risorsa e svolgere questa determinata azione”. I token possono scadere dopo n secondi, a quel punto si può richiedere un refresh token (dove permesso) o chiedere nuovamente all’utente di dare la propria autorizzazione all’app.

Modello

Un modello è un concetto fondante di molte librerie che si occupano di interazione col database, come in quella che usiamo noi, Sequelize. E' un modo, definito da ciascuna libreria, per creare la struttura delle risorse che andranno a definire il database (nel nostro caso i modelli sono 5: User, PerfectNight, Created, Saved e Upvoted). In parole molto semplici, è un modo per evitare di dover scrivere codice SQL: si può interagire con un database relazione definendo dei modelli, che la libreria poi converte in codice SQL e scrive sul database, e su questi modelli sono definiti dei metodi che sono, brutalmente, l'equivalente di scrivere query SQL per ricercare elementi all'interno del database.

Esempio pratico: sul modello user potremmo chiamare il metodo findOne, che prende come parametro l'opzione where, nella quale indichiamo che il nickname deve essere "mario". In javascript verrebbe una cosa come:

User.findOne({where:{nickname:"mario"}})

Questa funzione viene poi tradotta dalla libreria in codice SQL come:

SELECT *

FROM User

WHERE nickname = "mario

LIMIT 1

Il codice SQL viene fatto eseguire sul database e il risultato è poi restituito come risultato della chiamata findOne.

Ripeto, perché aiuta, è semplicemente un modo per non dover scrivere codice SQL all'interno del codice Javascript.

La struttura del backend

Il funzionamento del backend è, proprio come oAuth, racchiudibile in un banale flusso di esecuzione che si ripete all'infinito. Una volta che il server è stato avviato si mette in ascolto su una porta X:

- Quando arriva una richiesta tramite Internet a quella porta, "all'ingresso" del server c'è un oggetto chiamato **Router** che si occupa di vedere che tipo di richiesta è arrivata. Il suo comportamento è il seguente: legge il path della richiesta e il metodo della richiesta, e cerca questa combinazione metodo-path all'interno di tutte le combinazioni definite in app.js. Se trova un match, chiama in azione il singolo router specifico dove ha riscontrato un match, e torna in attesa "all'ingresso del server". Se non trova un match, invoca una route di default che corrisponde al nostro 404, la richiesta non è stata riconosciuta e torna in ascolto;
- Se è stato trovato un match ed è stato invocato uno dei mini router più specifici, il mini router si occupa di fare lo stesso match all'interno delle combinazioni metodo-path da lui definite: se trova un match, chiama la funzione del controller associata a quel match, altrimenti, nuovamente, 404.
- Il controller riceve la richiesta e implementa la logica necessaria a soddisfare quella richiesta: invia richieste ad altre api, consulta il database, svolge qualche operazione logico-matematica...