Express

Rev 4.4 del 28/02/2021

Express

http://expressjs.com

E' un modulo aggiuntivo di NodeJs che permette di gestire al meglio un server http.
Rispetto all'utilizzo diretto di Node presenta due vantaggi importanti :

- Presenza di un **dispatcher** integrato, che nella terminologia di Express si chiama **`router`** e che ha come compito quello di servire le richieste provenienti dal client.
- Disponibilità di molte librerie aggiuntive ad hoc per express (es express-session, body-parser, etc)

Esiste anche una utility **`express – generator`** che consente di creare automaticamente un web site completo basato su **`express`** e su **`jade`** che è un template per scrivere rapidamente pagine html che saranno poi convertite in html vero e proprio da jade stesso.
Abbastanza complicato senza peraltro apportare valori aggiunti effettivi.

Installazione di express

```
npm install express
```

Il primo progetto

```
var express = require('express');
var app = express();
```

express() restituisce il riferimento ad una applicazione web completa che può essere passata al metodo **`createServer()`** al posto della funzione di callback utilizzata nel caso di node js

```
const server = http.createServer(app);
server.listen(port, [ipAddress], callback())
```

Il metodo .listen() avvia un server sulla porta indicata. La funzione di callback viene richiamata in corrispondenza dell'avvio e serve per eventuali inizializzazioni.
Si potrebbe anche usare direttamente ***`app.listen()`*** che però non consente ulteriori configurazioni.

```
server.listen(1337, function () {
    var host = this.address().address;
    var port = this.address().port;
    console.log("Example app listening at http://%s:%s", host, port);
});

app.get('/', function (req, res, next) {
    res.send('This is the main page');
});

app.post('/page2.html', function (req, res, next) {
    res.send('This is page2 on post request');
});

app.use('/page3', function (req, res, next) {
    if (req.method.toLowerCase()=='get')
            res.send('page3 on get request');
    if (req.method.toLowerCase()=='post')
            res.send('page3 on post request');
});
```

La gestione delle routes

Nel linguaggio express i listener vengono indicati come **routes.**
Per ogni **route** occorre definire :

- **route method** (get, post, put, delete, etc)
- **route path** (la risorsa a cui rispondere)
- **route handler** (le funzioni di callback da eseguire come risposta alla richiesta)

In corrispondenza dell'arrivo di una richiesta, express scorre sequenzialmente l'elenco delle routes fermandosi automaticamente alla prima route che va a buon fine, cioè che corrisponde per il metodo (get, post) e per il nome della risorsa ("/api/elencoUser"). Quando trova una route che soddisfa questi parametri esegue il codice relativo a questa route.
Le routes vengono eseguite in sequenza, per cui è fondamentale l'ordine con cui vengono scritte.

app.get(resourse, callback(req, res, next)
Crea un listener per la gestione della risorsa GET indicata.

app.post(resourse, callback(req, res, next)
Crea un listener per la gestione della risorsa POST indicata

app.put(resourse, callback(req, res, next)
Crea un listener per la gestione della risorsa PUT indicata

```
app.route('/risorsa1')
```
consente di raccogliere in un unico contenitore metodi di risposta diversi (get, post, put, delete, etc) relativamente ad una medesima risorsa:

```
.get(function(req, res) {
    res.send('this is the get form');
})  // notare l'assenza del ;
.post(function(req, res) {
    res.send('processing the post request');
});
```

app.use(resource, callback(req, res, next) Crea un listener per la gestione della risorsa indicata. La funzione di callback viene eseguita **sempre**, indipendentemente dal metodo utilizzato per la richiesta (get, post, put, delete, etc)..

In tutti i metodi il parametro **resource** può assumere il valore **"*"** che ha il significato di "qualsiasi risorsa". In pratica il listener viene eseguito in corrispondenza di qualsiasi richiesta.

Il parametro next ed il codice middleware

Nel momento in cui una **route** va a buon fine, express la esegue ed interrompe la ricerca a meno che, al termine della procedura, non venga richiamato il metodo **next(),** quello passato come terzo parametro (opzionale) che è un puntatore ad una funzione interna di express che fa ripartire la scansione delle routes a partire dalla route successiva.

Le routes che utilizzano un next() finale, nel linguaggio di express sono indicate come **middleware (codice intermedio). Le funzioni middleware sono normalmente associate a routes aventi come risorsa "*", che dunque saranno eseguite in corrispondenza di ogni richiesta,** ad esempio per stampare dei log oppure per eseguire altre elaborazioni.

Una route in genere :

- **o esegue next()** (nel qual caso si parla di 'middleware')
- **oppure invia una risposta al client e termina la scansione**

```
app.use("*", function (req, res, next) {
   console.log(" -----> " + req.method + " : " + req.originalUrl);
   next();  });
```

Nel caso in cui nessuna **route** vada a buon fine si può aggiungere una route generica finale (da posizionarsi in coda a tutte le altre) per la gestione dell'errore:

```
app.use("*", function (req, res, next) {
    res.status(404);
    res.send("Sorry cant find that! ");
});
```

In realtà se la scansione arriva al termine del file senza eseguire nessuna end()/send(), express invierebbe automaticamente un messaggio di errore del tipo : **Cannot GET/POST resource**

http Response Methods

Nel caso di express, l'oggetto **respsonse** **espone alcuni metodi aggiuntivi** rispetto ai metodi standard esposti dall'oggetto httpResposnse di nodejs.

res.writeHead(200, header) res.end() // stessi di nodejs

res.**send**(str) Prima di inviare la risposta al client imposta automaticamente :

- **response-code = 200**
- **content-type** sulla base del tipo di oggetto passato a send (nel caso di una stringa sarà text/plain, nel caso di un json sarà application/json)
- Nel caso di un json, al contrario di **end()** provvede automaticamente a serializzare la risposta
- A differenza di **end()** trasmette automaticamente anche i cookies

res.**status**(404) Consente di impostare esplicitamente lo stato della risposta da inviare con .send()

res.**set**('**Content-Type**', '**text/html**') Consente di impostare esplicitamente il content-type della risposta da inviare con .send(). **Analogo al setHeader() di nodejs**.

res.sendStatus(code) Opposto al precedente. Invia una risposta fissa contenente il codice indicato e la sua stringa rappresentativa come body ("OK", "NOT FOUND"; "SERVER ERROR")

res.json() Invia uno stream jSon. Rispetto a send() serializza SEMPRE.
Nel caso di send() numeri e stringhe non vengono serializzati.

Inoltre, rispetto a .send(), consente di applicare due intestazioni specifiche che verranno passate a stringify per determinare alcune opzioni di serializzazione:

```
app.set('json replacer', replacer); // property transformation rules
app.set('json spaces', 2); // number of spaces for indentation
```

These two options are collected and passed to the JSON.stringify()method: since its signature loos like this: **JSON.stringify(object, replacer, space).**
Once this method is called, the **res.json()** method will then call **res.send()**.

res.sendFile(file) Legge e Invia il contenuto di un intero file (trasmesso in formato bytes stream).
Il **path** non può essere un percorso relativo ma deve necessariamente essere un percorso assoluto. Esisteva una vecchia versione **res.sendfile()** scritta tutto in minuscolo che accettava come parametro un percorso relativo, però deprecata.

La variabile **__dirname** rappresenta il percorso assoluto della cartella corrente (quella da cui è stato eseguito server.js)

```
app.use(function (req, res, next) {
       res.status(404);
       res.sendFile(__dirname + '/static/error.html')  });
```

In alternative è possibile utilizzare anche la seguente sintassi, che si aspetta come secondo parametro il percorso relative in cui cercare le risorse statiche

```
res.sendFile('error.html', {"root": './static'});
```

Durante l'apertura del file legge automaticamente anche il tipo del file e lo assegna alla property **Content-Type**

Altri Response Methods

res.redirect()	Redirect a request.
res.jsonp()	Send a JSON response with JSONP support.
res.download()	Prompt a file to be downloaded.
res.render()	Render a view template.

Regole di match delle routes: differenze tra app.use() e app.get()/app.post()

Mentre lo scopo di app.get(), app.post(), etc è quello ri **rispondere alle richieste del client**, lo scopo di app.use() è quello di eseguire codice **middleware**, che solitamente deve essere eseguito indipendentemente sia dal **metodo** di richiesta, sia dalla **risorsa** richiesta, per cui:

- `app.use()` risponde a su tutti i metodi di chiamata (get, post, put, delete, patch) mentre gli altri rispondono soltanto sul metodo indicato
- `app.use()` a differenza di app.get() e app.post() che eseguono un match perfetto sulla risorsa richiesta, **app.use non esegue un match esatto sulla risorsa richiesta, ma va a buon fine anche nel caso in cui la richiesta riguardi una qualsiasi sottorisorsa**. Ad esempio :

  `app.use('/apple', ..)` will match '/apple', '/apple/img', '/apple/img/news'
  In quest'ottica la richiesta `app.use('/')` va sempre a buon fine e, nel caso di app.use(), diventa quasi equivalente a "*". La piccolo differenza è spiegata nel paragrafo successivo.
  Tra le due è preferibile "/".
- `app.use()` consente di omettere la risorsa. Il valore di default infatti è **"/"** che significa tutte le risorse In app.get() e app.post() la risorsa NON PUO' ESSERE OMESSA. Al limite si usa un "*****".

Nota: Per tutte le routes, come 1° param, invece di una singola risorsa, si può passare un vettore di risorse.

Concetto di mount point

La risorsa per la quale è in ascolto un listener **middleware** è detta **mount point** della route.
Express, al momento della registrazione di una route middleware, va a modificare la proprietà **`req.url`** facendo in modo che contenga SOLTANTO la porzione di risorsa eccedente rispetto al mount point.
Al contempo aggiunge i seguenti campi:

`.originalUrl` contenente la url completa richiesta dal client
`.baseUrl` contenente il mount point su cui è in ascolto il listener
`.url` (equivale a **`path`**) rappresenta la parte rimanente tale per cui **`originalUrl = baseUrl + url`**

Esempio:

```
// Request : 'http://www.example.com/admin/files/news'
```

```
app.use('/admin/files', function(req, res, next) {
   console.log(req.baseUrl);       // /admin/files
   console.log(req.originalUrl);   // /admin/files/news
   console.log(req.url);           // /news  (equivalente a req.path)
   next();
});
```

Se si specifica come risorsa di ascolto `'/'`

req.orignalUrl	restituisce la risorsa completa
req.baseUrl	restituisce stringa vuota
req.url	**restituisce la risorsa completa**

Se invece si specifica come risorsa di ascolto `'*'`

req.orignalUrl	restituisce la risorsa completa
req.baseUrl	restituisce la risorsa completa
req.url	restituisce stringa vuota

A seconda dei casi cambia il valore della proprietà **url**
La non dichiarazione della risorsa di ascolto equivale a /

Dal momento che alcuni metodi (fra cui **express.static**() fanno uso della proprietà **url**, tra le due sintassi è preferibile la prima, quella dello **/**, in cui **url** restituisce la risorsa completa.

L'utilizzo dell'asterisco è pertanto sconsigliato, o per lo meno da usare solo nel caso degli application middleware in cui non interessa la url oppure con .get() e .post().

Built-in middleware and Third-party middleware

Oltre al cosiddetto **Application Middleware** (cioè le app.use() scritte manualmente dal programmatore) esistono anche delle funzioni middleware dette :

- **built-in middleware** cioè middleware integrato all'interno di express
- **third-party-middleware** cioè middleware disponibili in librerie esterne scritte appositamente per express, come ad esempio body-parser, cookie e session.

Questi middleware non ricevono esplicitamente i parametri `req, res e next`, ma accedono comunque alle istanze di questi oggetti e li possono leggere / scrivere.

A partire dalla versione 4, all'interno di express è rimasto un unico built-in middleware costituito dal metodo **express.static().** Moltissimi middleware sono invece disponibili sotto forma di libreria esterna

L'accesso alle risorse statiche del server

I file statici (immagini, css, js, favicon.ico) sono in genere gestiti all'interno di una cartella il cui nome può essere impostato a piacimento dal programmatore (es **public o static**) ed indicato ad express tramite il seguente middleware:

```
app.use('/', express.static('public') );          // oppure
app.use(express.static('./public') );             // oppure
app.use(express.static(__dirname + '/public') );  // oppure
app.use(express.static("."));          // cartella corrente
```

Cioè il metodo **express.static**() può ricevere indifferentemente un path assoluto oppure relativo.

Il metodo **express.static**() provvede a verificare se, all'interno delle cartella indicata, esiste la risorsa richiesta.

- Se esiste, la restituisce al client senza eseguire il metodo next.
- Se invece non esiste esegue il metodo **next()** e prosegue nella scansione delle routes dinamiche.

Si possono specificare in sequenza più middleware in ascolto su sottocartelle differenti.

```
app.use(express.static('./cartella1'));
app.use(express.static('./cartella2'));
```

Il secondo path viene aggiunto al primo senza sovrascriverlo.

Il primo parametro, di solito omesso, indica come sempre la risorsa di ascolto.
/ (oppure omettendo il parametrro) la route risponderà in corrispondenza di qualsiasi richiesta
/static la route risponderà **soltanto** per quelle risorse che cominciano con `/static` (prefisso che verrà però eliminato dal router come indicato nella seguente nota (3).

Note

1) Se il client richiede **/**, il listener statico cerca automaticamente il file **index.html**.
Se lo trova lo invia altrimenti richiama next().

2) Il metodo express.static() risponde soltanto in corrispondenza **delle richieste GET e HEAD**, e non in corrispondenza delle richieste **POST** che, secondo lo standard, sono riferite esclusivamente alle funzionalità di Insert / Create. Volendo è comunque possibile aggiungere una route personalizzata per la restituzione di risorse statiche in modalità POST:

```
app.post("/",function (req, res, next) {
    var pathName=url.parse(req.originalUrl).pathname;   (*)
    fs.readFile("static"+pathName, function (err,data) {
        if(err)
            next();
        else
            res.send(data);
    });
});
```

(*) url.parse() è necessario in quanto originalUrl potrebbe contenere eventuali parametri get.

3) Il primo parametro al solito ha il significato di **baseUrl**, mentre la risorsa richiesta dall'utente è memorizzata in forma completa all'interno di **originalUrl.**
Il campo utilizzato da express.static() per effetuare la ricerca è il campo **url**, ottenuto come differenza tra originalUrl e baseUrl.

- Utilizzando **baseUrl** = ***** **url** sarà vuoto per cui il listener, in corrispondenza di ogni richiesta successiva proveniente dalla pagina.html, continuerà sempre a restitire index.html !!
- Utilizzando **baseUrl** = **/,** la proprietà **url** coincide con la richiesta del client, per cui la risorsa viene individuata correttamente.

Sostanzialmente baseUrl rappresenta un punto di montaggio virtuale (**mount_path**) della cartella statica. Il client dovrà anteporre questo prefisso ad ogni richiesta; in corrispondenza di ogni richiesta il server provederà automaticamente a togliere tale prefisso

Si supponga di avere all'interno di static una sottocartella images. Per richiedere una immagine si potrà utilizzare una delle seguenti sintassi tutte equivalenti

```
1) app.use('/', express.static('static'));
   http://localhost:1337/images/desert.jpg
```

Il file **images/desert.jpg** viene cercato all'interno della cartella **./static**

2) `app.use('/images', express.static('static'));`
`http://localhost:1337/images/images/desert.jpg`
il listener andrà a cercare `images/desert.jpg` all'interno della cartella **`./static`**.
Il primo `images` viene automaticamente eliminato dalla url di ricerca (originalUrl - baseUrl)

3) `app.use('/images', express.static('static/images'));`
`http://localhost:1337/images/desert.jpg`
Il primo `images` viene automaticamente eliminato. Il listener andrà a cercare il file `desert.jpg` all'interno della cartella **`./static/images`**

Se index.html richiama una pagina2.html che si trova in una sottocartella **/pagine/pagina2.html**, quando poi pagina2.html richiama le sue risorse es
`<link rel="stylesheet" href="css/pagina2.css" />`)
il browser **automaticamente** antepone alla richiesta **/pagine** che era stato utilizzato per richiedere pagina2.html per cui la richiesta inviata al server sarà la seguente:
/pagine`/css/pagina2.css`
e, lato server, la ricerca dei file statici continua a funzionare senza dover aggiungere nessuna route statica

Lettura dei parametri GET

I parametri GET possono essere letti mediante **`req.query`**`.nomeParametro`

`http://localhost:1337/api/risorsa?nome=pippo`

```
app.use('/api/risorsa', function (req, res, next) {
    var nome = req.query.nome;
    res.send({"nome":nome, "residenza":"fossano"});
})
```

Attenzione però che **`request.query`** parsifica soltanto i parametri ricevuti in **url-encoded**. Si assume infatti che i parametri GET siano sempre passati in modalità url-encoded.

Se si vogliono intercettare i parametri GET passati in formato json occorre eseguire un parsing manuale:

```
app.use(function (req, res, next) {
    var _url = url.parse(req.url, false)
    var params = _url.query || "";
    params = decodeURIComponent(params);
    try {
        req["query"] = JSON.parse(params)
    }
    catch (error) {
        // i parametri erano url-encoded e sono già stati gestiti da express
    }
    next();   });
```

Lettura dei parametri POST

Per leggere i parametri POST occorre preventivamente installare il modulo body-parser
`npm install body-parser`

e quindi richiamarlo :
`var bodyParser = require('body-parser');`

La lettura dei parametri può poi essere eseguita attraverso le seguenti routes middleware che restituiscono i parametri stessi in formato json all'interno di **`req.body`**

```
app.use(bodyParser.urlencoded({limit:'50mb', extended:true}));
app.use(bodyParser.json({limit:'50mb', extended:true }));
```

Il **primo** middleware intercetta i parametri url-encoded**.**
Il **secondo** middleware intercetta i parametri scritti in formato json serializzato, restituendoli nella medesima property **request.body**

L'opzione **`limit`** serve per l'uload di immagini base64 all'interno dei parametri post
L'opzione `extended:false` consente di passare come parametri stringhe, vettori e anche oggetti json semplici codificati come object
L'opzione `extended:true` consente anche l'utilizzo di oggetti estesi (json object annidati)
Il default è true, però se si omette l'opzione extended il compilatore segnala un warning piuttosto fastidioso.

Parametri vettoriali

In caso di più parametri aventi tutti lo stesso nome vettoriale, come ad esempio
`nome=pippo&chkHobbies[ ]=sport&chkHobbies[ ]=cinema&chkHobbies[ ]=musica`
`req.query` e **`req.body`** restituisco entrambi, all'interno di un parametro **chkHobbies,** un vettore di stringhe, esattamente come avviene in PHP. In alternativa il client potrebbe passare come parametro un JSON contenente un campo vettoriale (soluzione più moderna).

Passaggio dei parametri come parte della risorsa

La tendenza attuale è quella di passare i parametri GET **non** accodandoli alla risorsa nel formato tradizionale della queryString **`?nome=valore`**, ma di inserirli direttamente all'interno della risorsa richiesta separandoli semplicemente tramite SLASH. Dovrà essere il codice server a gestire questi parametri in modo diverso, andando a leggerli direttamente all'interno della risorsa richiesta.

Supponendo di voler inviare al server la seguente richiesta GET:
`http://localhost:1337/api/risorsa?nome=pippo&eta=16`

i due parametri `nome` e `eta` possono essere direttamente accodati alla risorsa **in forma anonima:**

`http://localhost:1337/api/risorsa/pippo/16`

Il parametro passato in questo modo non sarà intercettabile all'interno della `queryString` tradizionale, ma occorrerà una apposita route scritta nel modo seguente:

```
app.get('/api/risorsa/:nome/:eta', function (req, res, next) {
    var nome = req.params.nome;
    var eta  = req.params.eta;
    res.send('good morning ' + nome );
```

Questa tecnica vale non solo per chiamate GET, **ma anche per tutte le altre chiamate**.
Infatti in qualsiasi tipo di chiamata è sempre possibile passare in modalità GET dei parametri aggiuntivi url-encoded o eventualmente espressi sotto forma di risorsa. Nel primo caso saranno disponibili all'interno di req.query, nel secondo caso saranno disponibili all'interno di req.params.

Questo approccio ha due **vantaggi**:

- il client può evitare l'utilizzo dei names
- E' disponibile un metodo aggiuntivo per la validazione preventiva dei parametri ricevuti

Ha però anche uno **svantaggio**: nel caso di utlizzo di app.get() oppure app.post() etc se mancano i parametri, il listener NON viene intercettato. Viene invece sempre intercettato nel caso di app.use(),

Validazione dei parametri passati come risorsa

I parametri passati come risorsa possono essere intercettati preventivamente e validati attraverso un metodo **`app.param()`** abbinato al metodo **`app.use()`** che definisce il nome dei parametri.

Nel momento in cui app.use( ) intercetta una richiesta e definisce il nome di una parametro passato come risorsa, automaticamente verifica se esiste una route di tipo **app.param**() associata a quale nome.
Se esiste la esegue **prima di procedere con la route corrente**.
Lo scopo di **app.param()** è quello di verificare la correttezza dei parametri ricevuti ed eventualmente interrompere il programma prima ancora dell'esecuzione della route effettiva identificata da app.use()

```
app.param('eta', function(req, res, next, valore) {
    console.log('\t doing name validations on ' + valore);
    if (valore!=13) next();
    else{
        var err = new Error('valore non consentito');
        next(err); }
})
```

app.param,() può essere scritto indifferentemente prima o dopo rispetto al corrispondente app.use().

La gestione degli errori

Nel caso in cui durante l'esecuzione si verifichi un qualsiasi tipo di errore, il router automaticamente richiama uno specifico middleware caratterizzato da una firma avente non **tre** parametri (req, res, next), ma **quattro** (**err**, req, res, next). Notare che il gestore degli errori deve necessariamente avere tutti e quattro parametri; il parametro **next** NON può essere omesso.

```
app.use(function(err, req, res, next) {
     console.log(err.stack)      // stack complete dell'errore // default
     console.log(err.message)    // ultimo messaggio in cima allo stack
     res.status(500).send(err.message)
});
```

Se questa route non è presente, il server si blocca **terminando** la sua esecuzione.

E' anche possibile generare nuovi errori ed interrompere l'esecuzione passando un apposito oggetto **err** al metodo next. Da usare ad esempio in caso di parametri mancanti o di valori non validi.

```
app.get('/api/data', function (req, res, next) {
    var nome = req.query.nome;
    if(!nome) {
        var err = new Error("Bad Request. Manca il parametro NOME");
        next(err);
    }
    else res.send("good morning " + nome + " " + eta );
});
```

La stringa passata a **new Error** sarà quella in cima allo stack e restituita da **err.message**
Il listener di errore dovrà essere posizionato alla fine dello script dopo tutti i listener di risposta.

Aggiunta di una funzione di log all'oggetto response

Nella maggior parte dei casi **il codice di errore è deciso dal chiamante**, che dovrebbe aggiungerlo all'oggetto **err** prima di richiamare **next(err)**. Per snellire al massimo il codice del chiamante, in fase di avvio si può assegnare all'oggetto **response** un nuvo metodo **log(err)** in modo che sia il chiamante medesimo a rispondere al client facendo allo stesso tempo, sulla stessa riga, il log() dell'errore.

```
app.response.log = function(err) {
     console.log(`********** Error ********* ${err.message}`)
}
```

chiamante:

```
if (err) res.status(500).send("Internal Error in Query Execution").log(err)
```

I server CRUD (Create Read Update Delete)

Le operazioni base che si possono eseguire sui record di un database relazionale sono tipicamente quattro: Create/Insert, Read (Select), Update e Delete operazioni solitamente indicate con l'acronimo CRUD.

I principali metodi con cui un cient http può accedere ad una risorsa sono essenzialmente anche quattro: GET, POST, PUT e DELETE.

Un server CRUD è un http server che, facendo riferimento sempre alla medesima risorsa, associa i quattro metodi HTTP con le quatto operazioni CRUD. Cioè il medesimo servizio si comporta in modo diverso a seconda del metodo con cui viene effettuata la chiamata:

GET -> Lettura di un record
POST -> Inserimento di un nuovo record passato come parametro
DELETE -> Cancellazione di un record il cui ID di solito viene passato come risorsa */people/13*
PUT/PATCH -> Aggiornamento di un record il cui ID di solito viene passato come risorsa */people/13*

Notare che questo principio è in contrasto con quello che è l'utilizzo comune dei metodi HTTP.
Molto spesso infatti il metodo GET viene utilizzato per eseguire qualsiasi tipo di interazione con il server. Ad esempio, in una applicazione Web, per l'inserimento di un nuovo record si può comunemente eseguire una richiesta di tipo GET su un URI del tipo:

```
http://www.myapp.com/addCustomer?name=Rossi
```

Questo utilizzo del metodo GET ad ampio spettro non è però conforme ai principi **REST**, secondo i quali il metodo GET dovrebbe servire **esclusivamente** per accedere ai dati di una risorsa.

Richiami sugli HTTP Request Method

Sono complessivamente nove. RFC 2616 (HTTP/1.1)

Oltre ai tre metodi base (GET, POST e HEAD), in HTTP 1.1 sono stati aggiunti altri **sei Request Method** che consentono di specificare meglio l'azione che deve essere eseguita su una certa risorsa. Tranne POST (e in parte PATCH) tutti i metodi sono di tipo **`idempotent`**, cioè se **la stessa richiesta** viene inviata più volte produce sempre lo stesso risultato senza modificare lo stato del server (che è uno dei pilastri di HTTP). Cioè se il metodo viene richiamato una sola o più volte consecutive, il risultato è sempre lo stesso. Aspetto molto importante nell'ambito della fault-tolerance. Se un comando idempotent va in timeout (perché si perde la risposta del server) lo posso rieseguire una seconda volta senza alcun rischio. POST non è idempotent perché ad ogni richiesta aggiunge un nuovo record fariando variare lo stato del server. DELETE è idempotent perché alla prima esecuzione cancella il record, mentre nelle esecuzioni successive non esegue alcune azione, per cui il risultato finale è indipendente dal numero di richiami.

`GET`	The `GET` method requests a representation of the specified resource. **Requests using `GET` should only retrieve data**.
`HEAD`	The `HEAD` method asks for a response identical to that of a `GET` request, but without the response body.
`POST`	**The `POST` method is used to submit an entity to the specified resource**, often causing a change in state or side effects on the server.
`PUT`	The `PUT` method **replaces all current representations of the target resource with the request payload**. // Sostituisce l'intera risorsa oppure crea una nuova risorsa nel caso in cui il record ricercato non esista (es opzione **upsert** di mongodb)
`PATCH`	The `PATCH` method is used to **apply partial modifications to a resource**.
`DELETE`	The `DELETE` method **deletes the specified resource**.
`OPTIONS`	The `OPTIONS` method is used to describe the communication options for the target resource.
`TRACE`	The `TRACE` method performs a message loop-back test along the path to the target resource.
`CONNECT`	The `CONNECT` method establishes a tunnel to the server identified by the target resource.

Gestione delle richieste extra domain (CORS)

Quando si richiede una pagina ad un server, all'interno della pagina medesima viene sempre memorizzato il dominio del server dal quale la pagina è stata scaricata.
CORS (**Cross-Origin Resource Sharing**) è un meccanismo per controllare **lato server** se le richieste ajax provenienti dal client sono inviate da **pagine** appartenenti o meno allo stesso domino del server ajax.

Capita infatti spesso di dover richiedere dei dati ad un server diverso rispetto a quello da cui è stata scaricata la pagina. Ad esempio quando si utilizza un server **Angular** (pubblicato ad esempio su firebase) le cui pagine inviano richieste Ajax ad un server dati esterno (ad esempio heroku).

Quando il server riceve una richiesta ajax, controlla all'interno della richiesta il dominio dei appartenenza della pagina che sta inviando la richiesta. Se il dominio di appartenenza della pagina è lo stesso del server invia I dati, altrimenti se la pagina proviene da un altro server non accetta la richiesta e restituisce un codice di errore all'interno di una apposita http header in corrispondenza della quale il browser solleva una eccezione del tipo "**`CORS policy error`**".

Con il termine ***`cross-origin HTTP request`*** si intende la richiesta da parte dell'applicazione di una risorsa che differisce dall'origine corrente per almeno une delle seguenti condizioni:

- **dominio**
- **protocollo** di accesso
- **porta** di ascolto.

Fra queste tre problematiche la più frequente è sempre la prima, cioè quella legato al dominio.
Per questo le problematiche CORS vengono spesso iassociate al termine **`extra-domain request`**.

Come detto un http server per default opera in **`restricted mode`**, cioè NON risponde a richieste extra-domain. E' compito del programmatore del server decidere se abilitare o meno il server a rispondere alle richieste CORS. In un server express per abilitare le richieste CORS si può utilizzare il seguente codice:

```
app.use("/", function(req, res, next) {
    // res.setHeader("Access-Control-Allow-Origin", "*")
    res.setHeader("Access-Control-Allow-Headers", "*");
    res.setHeader("Access-Control-Allow-Methods", "*");
    res.setHeader('Access-Control-Allow-Credentials', true);
    next();
});
```

`res.setHeader("Access-Control-Allow-Origin", "*")`
rappresenta l'intestazione "storica" per l'abilitazione CORS, cui di dichiara di accettare richieste provenienti da qualsiasi origine

`res.setHeader('Access-Control-Allow-Headers', "*");`
Questa intestazione serve SOLO se la richiesta ha un'intestazione Access-Control-Request-Headers e indica quali headers la response è in grado di fornire. Si possono passare più headers separate da virgola

`res.setHeader('Access-Control-Allow-Methods', "*");`
abilita le richieste CORS su tutti gli HTTP methods. Per default sono ablitati solo GET e POST

`res.setHeader('Access-Control-Allow-Credentials', true);`
abilita la trsmissione dei cookie all'interno delle richieste CORS.
E' equivalente al campo **`credentials`** della libreria CORS

In realtà la prima di queste direttive sembra andare in conflitto con l'ultima. **Solitamente in express le headers precedenti vengono sostituite con un apposito middleware `cors()` indicato di seguito**:

```
const whitelist = ["http://localhost:8080", "https://localhost:1337",
                   "http://192.168.137.1:8080", "https://192.168.137.1:1337"];
const corsOptions = {
    origin: function(origin, callback) {
        if (!origin)
            return callback(null, true);
        if (whitelist.indexOf(origin) === -1) {
            var msg = 'The CORS policy for this site does not ' +
                'allow access from the specified Origin.';
            return callback(new Error(msg), false);
        }
        else
            return callback(null, true);
    },
    credentials: true
};
app.use("/", cors(corsOptions));
```

All'interno del vettore **whitelist** si imposta un elenco di domini validi. Se la pagina che sta inviando la richiesta CORS è stata scaricata da uno di questi domini il server accetterà la richiesta ed invierà i dati.

Se si desidera accettare richieste da qualunque sorgente, si potrebbe impostare il seguente middleware:

```
const corsOptions = {
    origin: function(origin, callback) {
          return callback(null, true);
    },
    credentials: true
};
app.use("/", cors(corsOptions));
```

Ad esemp le API di **randomuser.me** possono essere richiamate da qualsiasi dominio (solo tramite **https)**

Viceversa molte altre API non forniscono servizio CORS e, in questi casi non si può fare nulla.

- Se si invia la richiesta dalla barra di navigazione del browser tutto ok.
- Se invece si invia una richiesta Ajax viene fuori un errore CORS.

E' comunque possibile aggirare il problema inviando la richiesta non dal client ma dal server. Il client richiede i dati al proprio server, il quale invia una richiesta HTTP alla API esterna e poi ritorna il risultato al client:

```
var request = require('request');
request("https://api.xxxxx.com", function(error,responseStatus,data){
   if(!error)   res.send(JSON.stringify(data))
});
```

Nel febbraio 2020 l'oggetto **request** è stato deprecato per ragioni di sicurezza.
Tra le alternative si consiglia la libreria **axios** (npmjs.com)

Gestione delle Variabili SESSION

Le variabili session sono associate ad una sessione utente e consentono di memorizzare delle informazioni relative a quel singolo utente. La loro gestione è basata sul middleware **session**

```
var session = require('express-session');
app.use(session({
    secret: "myKeyword",
    name: "sessionId",
```

```
    // proprietà legate allo Store
    resave: false,
    saveUninitialized: false,
    cookie: {
        secure: false,      // true per accessi https
        maxAge: 60000       // durata in msec
    }
}));
```

Questo middleware sostanzialmente genera un nuovo sessionID e lo memorizza all'interno di un cookie.

Il primo parametro (secret) rappresenta la chiave da utilizzare per l'autenticazione HMAC, chiave che verrà utilizzata come chiave di hash per la cifratura irreversibile di sessionID

Il secondo parametro (name) rappresenta il nome da assegnare al cookie in cui salvare il sessionID
// The default value is **`'connect.sid'`**.

Una volta creato ed inizializzato l'oggetto `session`, le variabili session vengono memorizzate nella RAM del server di back end in modo trasparente. E' possibile rendere persistenti le variabili SESSION salvandole su un file oppure all'interno di un DB (mongo). In questo modo non vanno perse in caso di riavvio del server e soprattutto sono più facilmente gestibili nel caso di server distribuiti.

Sintassi per l'utilizzo delle variabili session

In ogni caso la variabili SESSION risultano accessibili al programmatore sempre allo stesso modo, cioè tramite l'oggetto **`session`** dell'oggetto **`request`** :

```
request.session
```

Creazione di una variabie / Assegnazione di un valore

```
req.session.name = 'Napoleon';
req.session["nome composto"] = 'Napoleon';
```

Lettura di un valore:

```
var name = req.session.name;
var nome_composto = req.session["nome composto"];
```

Cancellazione di una variabile

```
delete req.session.name
delete req.session["nome composto"];
```

Rimozione dell'oggetto session

```
req.session.destroy();
```

Questo metodo accetta come parametro una eventuale funzione di callback

```
req.session.destroy(function() {
    res.send('Session deleted') });
```

Tempo di durata della sessione

```
// This user won't have to log in for a year
req.session.cookie.maxAge = 365 * 24 * 60 * 60 * 1000;
// This user should log in again after restarting the browser
req.session.cookie.expires = false;
```

I cookies con durata 0 (oppure `expires:false`) non vengono memorizzati dal browser e dunque vengono rimossi SOLO quando il browser viene chiuso. Per far decadere subito il cookie occorre impostare un tempo piccolissimo (min 1 msec).

Upload di un file tramite Ajax

Per eseguire l'upload di un file è disponibile la comodissima libreria **express-fileupload** che intercetta il file lato server e lo restituisce all'interno di **req.files**

```
const fileupload = require('express-fileupload');

app.use(fileupload({
    limits: { fileSize: 50 * 1024 * 1024 },
}));
```

Il middleware **fileupload** intercetta i parametri passati in modalità FormData e li restituisce all'interno dell'oggetto **req.files** sotto forma di vettore enumerativo di json, un record per ogni file. **Attenzione: che se il file è uno solo invece di un vettore enumerativo viene restituito un singolo json**

Lato Client

Il lato client è il solito. Cioè :

1. Lato HTML si utilizza un semplice controllo <**input type="file**">

   ```
   Scegli un file : <input type="file" id="txtFile" multiple ><br>
   <input type="button" value="Upload" id="btnInvia">
   ```

2. Prima dell'invio occorre trasformare il vettore enumerativo **files** restituito da **txtFile** in un vettore associativo di tipo **Form-Data** in cui ogni property assume la forma di un vettore enumerativo di valori:

   ```
   var files = $('#txtFile').prop('files');

   var formData = new FormData();
   for (let i=0; i<files.length; i++)
       formData.append('txtFile', files[i]);

   var request = inviaRichiestaMultipart("post", "/api/upload", formData);
   ```

3. Inviare l'oggetto **files** come flusso binario senza aggiungere controlli / serializzazioni.

   ```
   data:parameters,
   contentType:false,
   processData:false,
   ```

La conversione in FormData deve essere eseguita anche in caso di file singolo.
A differenza di apache, nel caso di express **le parentesi quadre dopo il nome della variabile da trasmettere sono del tutto irrilevanti** (meglio ometterle !)

Lato Server: File Singolo

```
let file = req.files.txtFile;
```

file è un json contenente i seguenti campi:

```
file.name: client filename - "car.jpg"
file.mimetype: The mimetype of your file
file.size: The dimension of your file
```

```
file.data: A buffer representation of your file
file.truncated: A boolean that represents if the file is over the size limit
file.mv: A function to move the file elsewhere on your server
file.md5: A function that returns an MD5 checksum of the uploaded file
```

Le utlime 4 voci vengono aggiunte da **fileupload** compreso il comodissimo **mv** per lo storage del file.

Esempio completo File Singolo

```
app.post('/api/upload', function (req, res, next) {
    if (!req.files || Object.keys(req.files).length == 0)
           res.status(400).send('No files were uploaded');
    else{
        let _file = req.files.txtFile;
       _file.mv(__dirname + '/static/upload/' + _file.name, function(err) {
                if (err)
                     res.status(500).json(err.message);
                else
                     res.json("ok");
       })
    }
});
```

Se il file è già esistente viene semplicemente sovrascritto.
Il listener funziona allo stesso identico modo anche utilizzando una chiamata **put**

Esempio completo File Multipli

```
app.put('/api/upload', function (req, res, next) {
    if (!req.files || Object.keys(req.files).length == 0)
        res.status(400).send('No files were uploaded');
    else
    {
        let  files = req.files.txtFile;
        async.forEach(_files,
            function(file, callback ){
                file.mv(__dirname + '/static/upload/' + file.name,
                    function(err) {
                       callback(err)
                    })
            },
            function(err) {
                res.set("content-type", "application/json;charset=utf-8")
                if (err)
                    res.status(500).json(err.message);
                else
                    res.json("ok");
            }
        );
    }
});
```

La libreria socket.io e l'utilizzo dei web socket

L'API WebSocket è stata standardizzata dal **W3C** e il protocollo WebSocket è stato standardizzato dall'IETF come RFC 6455. I web socket consentono la creazione di una connessione TCP all'interno di una connessione HTTP in modo da consentire al server di poter inviare al client messaggi asincorni come ad es le notifiche. Due notevoli vantaggi:

- Il fatto che il server possa inviare notifiche al client senza che sia il client a richiederle
- La connessione TCP, lato server, "passa" sempre attraverso la porta 80 del web server, **il che costituisce un vantaggio per gli ambienti che, tramite firewall, bloccano l'utilizzo di porte non standard.**

Per prima cosa occorre creare ed avviare un server HTTP standard :

```
const http = require ("http");
const express = require('express');
const app = express();
const server = http.createServer(app);
server.listen(1337, function () {
    console.log("Server listening on port 1337");
});
```

Occorre quindi creare una istanza della clase **socket.io** passando al costruttore l'oggetto http server su cui appoggiarsi.

```
const io = require('socket.io')(server);
```

Lato client : invio della richiesta di connessione

La medesima libreria socket.io è disponibile lato client sul sito ufficiale **socket.io** o tramite CDN
Questa libreria rende utilizzabile lato client un oggetto javascript **io** contenente tutti i metodi / eventi necessari per la comunicazione tramite web socket con il server precedente.

La connessione attraverso il web socket deve essere aperta dal client nel modo seguente:

```
var socket = io.connect();
```

che invia una richiesta di connessione al server http da cui è stata scaricata la pagina corrente.
Restituisce il socket di connessione (esattamente come nel caso della libreria net).
L'oggetto socket dispone di un primo evento generato in corrispondenza della risposta del server:

```
socket.on('connect', function(){
     console.log("connessione ok");
});
```

Lato server: accettazione della connessione

In corrispondenza di una richiesta di connessione da parte di un client, lato server viene generato un evento **connection** a cui viene iniettato come parametro il **socket** contenente tutte le informazioni relative al client che ha richiesto la connessione. Il socket di solito viene salvato in una variabile globale o in un vettore globale di socket nel caso di gestione di molteplici connessioni.

```
io.on('connection', function (_socket) {
     console.log(' User ' + socket.id + ' connected!');
     socket = _socket;
});
```

Il campo **socket.id** contiene ID univoco assegnato dal server in corrispondenza di ogni richiesta di connessione. E' il campo che consente di identificare tutti i vari messaggi sucessivi.

Scambio di dati attraverso il `socket`

A questo punto, stabilita la connessione, sia il client che il server possono inviare dei dati sulla connessione mediante il metodo **`emit()`** che presenta due parametri entrambi di tipo stringa.

`socket.emit(dataName, data)`;

Il primo parametro **`dataName`** è un identificativo che identifica univomanete il dato.
Il secondo parametro **`data`** rappresenta il dato vero e proprio

Sull'host remoto, in corrispondenza di ogni emit(), si verfica un evento differente per ogni **`dataName`** inviato dal mittente. I vari eventi presentano lo stesso **`dataName`** definito dal mittente ed hanno come callback una funzione alla quale vengono iniettati i dati ricevuti.

`socket.on(dataName, function (data) { }`

Messaggi di broadcast

`socket.broadcast.emit(dataName, data);` Questo metodo consente al server di inviare dei dati in broadcast a tutti gli host attualmente connessi, con esclusione di quello identificato dal socket corrente

`io.sockets.emit(datatName, data)`; Metodo statico che consente al server di inviare un messaggio a tutti gli host attualmente connessi (mittente compreso).

Chiusura della connessione

La richiesta di chiusura della connessione può essere eseguita indifferentemente in qualsiasi momento sia dal client che dal server tramite il richiamo del metodo

```
socket.disconnect();
```

Normalmente comunque è il browser che invia la richiesta di disconnessione, che viene inviata automaticamente in corrispondenza della chiusura della scheda di navigazione.

In corrispondenza del richiamo del metodo **`socket.disconnect()`** viene automaticamente generato sia sul client sia sul server il seguente evento:

```
socket.on('disconnect', function(){});
```

Il client può eventualmente anche richiedere la chiusura della connessione al server:
client:

```
io.emit('end');
```

server:

```
socket.on('end', function (){
    socket.disconnect();
});
```

Passaggio dei parametri in fase di connessione

Al momento del **connect** il client può passare al server eventuali parametri:

Client:

```
let socket = io.connect("", {"query":`name=${username}`});
```

Server:

```
let username = socket.handshake.query.name;
```

Timeout

In caso di mancanza di interazione con la pagina, il client di tanto in tanto invia dei **ping** al server il quale risponde con i cosiddetti **pong**. In fase di istanza del server, è possible definire I due parametri in ms:

```
const io = require('socket.io')(server,
{
  pingInterval: 25000,  // default 25 sec
  pingTimeout:   5000,  // default  5 sec
});
```

`pingInterval` indica la frequanza di tempo con cui il client deve inviare i ping al server
`pingTimeout` indica il timeout di attesa del pong da parte del client

Nel momento in cui un client si connette al server, il server invia automaticamente al client questi due parametri definendo in pratica il funzionamento del client.
Il client di conseguenza instazia all'interno del web api un nuovo thread che in corrispondenza di ogni **`pingInterval`** invia al server un messaggio di ping. **Se lo script va in errore, il thred termina termindo anche tutti i thread posizionati nelle webapi, compreso quello di invio dei ping.**

Se dopo la connessione il thread va subito in errore, il timeout effettivo di disconnessione sarà dato dalla somma di **`pingInterval`** (attesa per invio ping) **`+ pingTimeout`** (attesa risposta server)

Attenzione che **ping** e **pong** sono nomi riservati che rappresentano chiavi predefinite nella cominicazione tra client e server e dunque non possono essere utilizzati come chiave nei normali messaggi utente.

Creazione di un server HTTPs

In caso di servizi protetti, la trasmissione del token deve assolutamente avvenire all'interno di una trasmissione HTTPs in modo che il token non possa in alcun modo essere intercettato.

Il protocollo HTTPs (basato su TLS/SSL) prevede un tipo di autenticazione detta ***unilaterale*** in cui solo il server è autenticato (il client conosce l'identità del server), ma non viceversa (il client rimane anonimo e non autenticato). Il client valida il certificato del server controllando la firma digitale dei certificati del server, verificando che questa sia valida e riconosciuta da una *Certificate Authority* conosciuta utilizzando una cifratura a chiave pubblica.

Il passo necessario per poter creare un server https è quello di creare una **coppia di chiavi RSA**:

- una **chiave privata** da tenere memorizzata sul server
- una chiave pubblica da distribuire tramite **Certificato Digitale** controfirmato digitalmente da una Certification Authority.

OpenSSL

OpenSSL è una applicazione che consente di creare certificati digitali controfirmati oppure self-signed.
Il sito ufficiale ***openssl.org*** fornisce e mantiene soltanto la versione Linux.
Al momento (novembre 2018) l'ultima versione è la v ***1.1.1d*** .
Per quanto riguarda le versioni windows si consigliano i seguenti link:

https://slproweb.com/products/Win32OpenSSL.html (con eseguibile di installazione)
sourceforge.net che fornisce una più flessibile versione .zip che può essere 'copiato' in qualsiasi cartella

Per ogni versione sono disponibili due distribuzioni:

- quella ***full*** (circa 43MB) raccomandata per gli sviluppatori (*indispensabile per la creazione dei certificati*) e che necessita di un file di configurazione distribuito insieme al pacchetto
- quella ***light*** (circa 3MB) raccomandata per i semplici utilizzatori e che non necessita del file di configurazione. Ad esempio la libreria di node OPCUA client per il funzionamento richiede la presenza di openssl nel profilo utente, ed è *sufficiente* la versione light

openssl in realtà viene già installato insieme a **git**, per cui se si ha git installato sul pc non c'è bisogno di ulteriori installazioni. In tutti i casi occorre però creare le seguenti variabili di ambiente (dove ***OPENSSL_CONF*** esiste solo nella versione full) :

```
PATH          ->   F:\programmi\git\usr\bin
OPENSSL_CONF  ->   F:\programmi\git\usr\ssl\openssl.cnf
```

A questo punto da qualsiasi cartella di lavoro si può lanciare il comando sotto indicato.
Se non si impostano le variabili di ambiente si può lanciare il comando direttamente dalla cartella BIN di openssl. E' anche possibile digitando solo openssl aprire un terminale all'interno del quale si possono inviare gli stessi comandi omettendo la voce iniziale openssl.

openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout privateKey.pem -out certificate.pem

I formati di memorizzazione dei certificati x.509

.pem indica una chiave privata o un certificato x.509 memorizzati in formato testuale base64
.der indica una chiave privata o un certificato x.509 memorizzati in formato binario
.key estensione generica per una chiave privata, che può essere memorizzata sia in pem che in der
.crt estensione generica per un Certificto Digitale, che può essere memorizzato sia in pem che in der
.csr o . req indicano un `Certificate Signing Request` cioè un Certificato Digitale da inviare ad una Certification Authority per l'apposizione della firma digitale.

Esistono diversi siti di tipo **`SSL Certificate Decoder`** che consentono di visualizzare il contenuto di un certificato PEM o DER

Significato dei parametri

https://www.digitalocean.com/community/tutorials/how-to-create-a-self-signed-ssl-certificate-for-apache-in-ubuntu-16-04

- **req**: This subcommand specifies that we want to use X.509 certificate signing request (CSR) management. The "X.509" is a public key infrastructure standard that SSL and TLS adheres to for its key and certificate management. We want to create a new X.509 cert, so we are using this subcommand.
- **-x509**: This further modifies the previous subcommand by telling the utility that *we want to make a self-signed certificate* instead of generating a certificate signing request, as would normally happen.
- **-nodes**: This tells OpenSSL to skip the option to secure our certificate with a passphrase. We need Apache to be able to read the file, without user intervention, when the server starts up.
  A passphrase would prevent this from happening because we would have to enter it after every restart.
- **-days 365**: This option sets the length of time that the certificate will be considered valid. We set it for one year
- **-newkey rsa:2048**: This specifies that we want to generate a new certificate and a new key at the same time. We did not create the key that is required to sign the certificate in a previous step, so we need to create it along with the certificate. The `rsa:2048` portion tells it to make an RSA key that is 2048 bits long.
- **-keyout**: This line tells OpenSSL where to place the generated private key file that we are creating.
- **-out**: This tells OpenSSL where to place the certificate that we are creating.

As we stated above, these options will create both a key file and a certificate. We will be asked a few questions about our server in order to embed the information correctly in the certificate. Fill out the prompts appropriately. The most important line is the one that requests the Common Name (e.g. server FQDN Fully Qualified Domain Name or **YOUR name**). You need to enter the domain name associated with your server or, more likely, your server's public IP address.

```
Output
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:New York
Locality Name (eg, city) []:New York City
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Bouncy Castles, Inc.
Organizational Unit Name (eg, section) []:Ministry of Water Slides
Common Name (e.g. server FQDN or YOUR name) []:server_IP_address
Email Address []:admin@your_domain.com
```

Creazione di un server HTTPs basato su express

```
const privateKey  = fs.readFileSync("keys/privateKey.pem", "utf8");
const certificate = fs.readFileSync("keys/certificate.pem", "utf8");
const credentials = { "key": privateKey, "cert": certificate };

var express = require('express');
var app = express();

var httpServer = http.createServer(app);
httpServer.listen(8080);

var httpsServer = https.createServer(credentials, app);
httpsServer.listen(8443);
console.log("server in ascolto sulle porte 8080 e 8443");
```

I due server http e https possono essere in esecuzione contemporanea consentendo accesso http e https

Quando si usa https ricordarsi di aggiungere **`secure=true`** al cookie.

Hotspot mobile

Windows 10 fornisce un comodissimo servizio di **Hotspot mobile** che consente di trasforma il modulo WiFi del PC in un Access Point / DHCP server verso l'esterno. Per i PC non dotati di moduli WiFi si può montare sul PC una antennina USB che consente sostanzialmente di attivare il WiFi

Una volta terminato un progetto e avviato il server su un PC locale, per renderlo fruibile su uno smartphone è sufficiente abilitare il suddetto **`hotspot mobile`**

Configurazione / attivazione Hotspot mobile:

- Start / **Impostazioni** / Rete e Internet / **Hotspot mobile** (menù di sinistra)
- Condividi la connessione Internet con altri dispositivi / **Attiva**
- Eventualmente Modificare / Nome Rete e Password

L'hotspot si crea lui una nuova rete (typ **192.168.137.0**) riservando il primo indirizzo per se stesso (**192.168.137.1**) ed assegnando gli altri indirizzi via DHCP ai dispositivi che ne fanno richiesta.

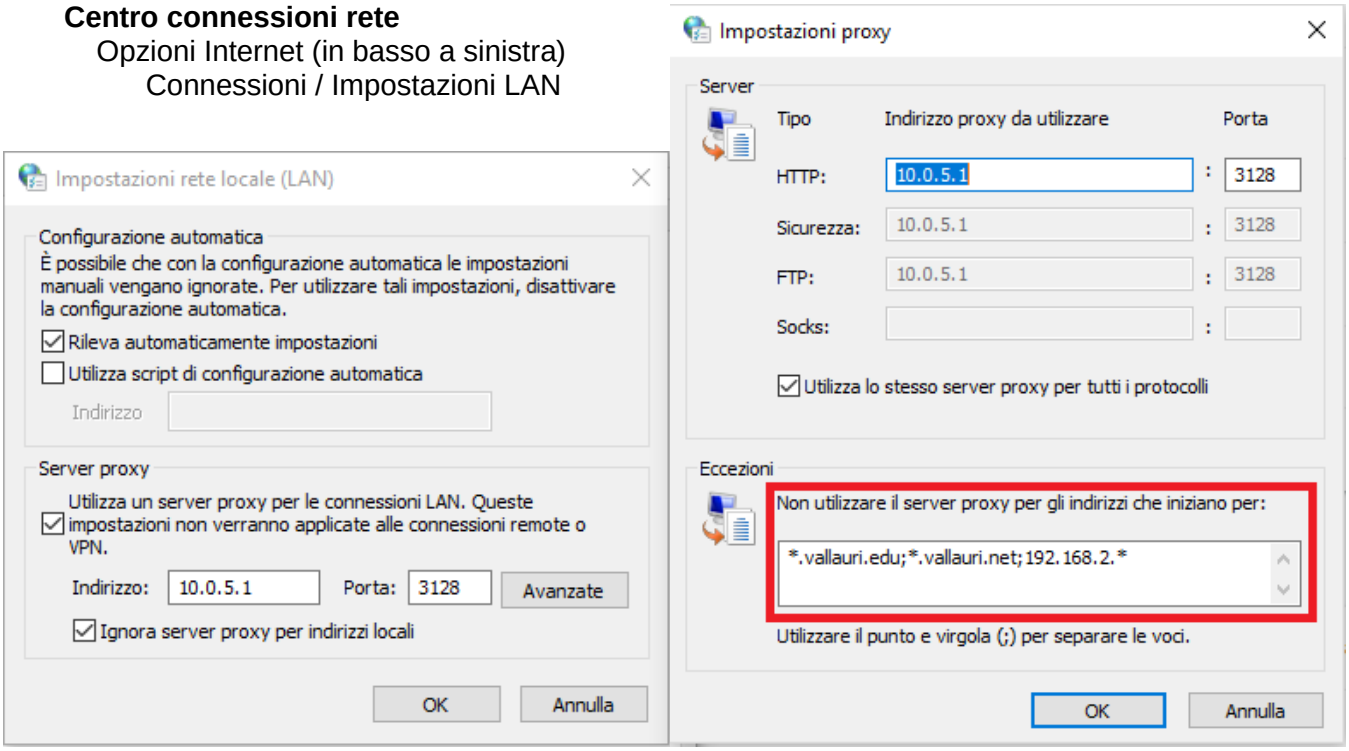
Dopo di che per accedere ad un server in esecuzione sul PC occorre:

- avvaire il server in ascolto sull'indirizzo **192.168.137.1** (oppure senza specificare nessun indirizzo IP, quindi su tutte le interfacce)
- Aprire il browser sul dispositivo mobile e puntare all'indirizzo **192.168.137.1:1337**

E' consentito un max di **8 device** contemporanei

Impostazione / Eliminazione del Proxy da Pannello di Controllo

Pannello di controllo
Centro connessioni rete
Opzioni Internet (in basso a sinistra)
Connessioni / Impostazioni LAN



Hosting di una applicazione su HEROKU.COM

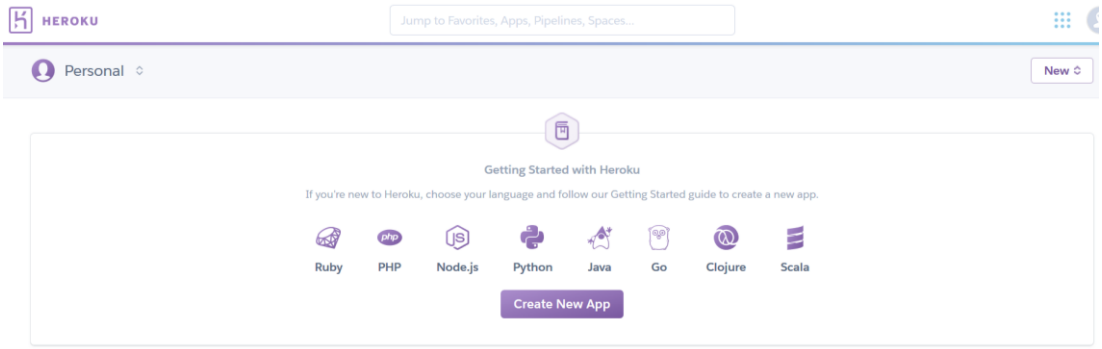
heroku.com è un provider in capo a **Salesforce.com** impresa statunitense di cloud computing con sede a San Francisco, California e operativa in 36 paesi del mondo. Provider altamente professionale su cui si appoggia anche Facebook. Si parte da un servizio base gratuito a servizi che arrivano fino 50.000 € / mese, per sistemi dedicati con 16G di RAM e 100 dynos (https://www.heroku.com/**pricing**)

Con il termine **web dyno** (https://www.heroku.com**/dynos**) heroku indica il numero di **web workers** attivi all'interno di una app. Quando arriva una web request, questa viene 'accodata' ad un worker scelto casualmente, il quale esegue il job indicato interagendo eventualmente con il DB e restituendo i dati richiesti. Maggiore è il numero di worker attivi parallelamente è più reattivo sarà il server.

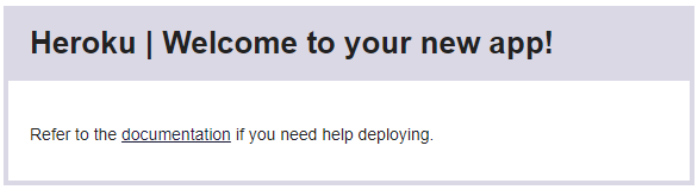
La versione gratuita fornisce 550 ore di listening mensili su un singolo dyno (23 giorni).
Il dyno va automaticamente in stand-by dopo 30 minuti di inattività
Prerequisiti : **nodejs** e **git**

❖ Registrazione

- Registrarsi su ***heroku.com*** creando un account personale costituito da email + password
- Scegliere come linguaggio "NODEJS"



- Cliccare "***Create New App***"
- Assegnare un nome univoco alla App (che deve essere scritto in minuscolo senza spazi).
  Ad es ***roberto-mana-app*** che sarà il nome con cui la app sarà esposta su heroku tramite la url:
  https://roberto-mana-app.herokuapp.com/
  E' comunque possibile rinominare la app in qualunque momento
- In corrispondenza dell'OK viene creata una app vuota associata alla url indicata che si presenta nel modo seguente:



  La app sarà accessibile via http alla url sopra indicata e potrà essere aggiornata tramite **git** all'indirizzo ***git.heroku.com/roberto-mana-app.git***

❖ Installazione di Heroku CLI

https://devcenter.heroku.com/articles/heroku-cli
https://devcenter.heroku.com/articles/getting-started-with-nodejs#set-up

Heroku CLI consente di gestire un terminale di comunicazione con Heroku. Può essere installata in qualunque path. La pagina di installazione contiene una guida sintetica con i principali comandi da utilizzare per crere e testare l'applicazione

❖ Login

- Creare in qualunque posizione una propria cartella di lavoro e copiare all'interno l'applicazione da caricare su Heroku.
- Eseguire ***heroku login*** per abilitare Heroku CLI a comunicare con il server.
  Il login viene effettuato tramite l'apertura di una pagina web.

in alternativa, si possono impostare le credenziali direttamente nella da linea di comando (heroku login –i). Il login ha un timeout di alcuni giorni, dopo di che, in caso di inutilizzo, deve essere rifatto.

❖ Adattamento dell'applicazione

- L'applicazione verrà avviata sulla porta 80, con esclusione di qualunque altra porta.
  Occorre pertanto modificare la definizione della porta di ascolto nel modo seguente :
  `const PORT = process.env.PORT || 1337`
- Per poter eseguire l'upload è **indispensabile** la presenza del file **package.json** all'interno della cartella principale che indicherà ad heroku tutte le dipendenze del nostro progetto, ciascuna con la propria versione.
- Il file package.json potrebbe essere 'copiato' da qualche parte e poi adattato, però si rischia di perdere qualche pezzo e, soprattutto, ad ogni nuova aggiunta occorrerebbe aggiornarlo manualmente. Molto meglio creare un "vero progetto" creando manualmente la cartella node_modules ed installando in locale tutte le dipendenze necessarie. A questo punto il file package.json può essere creato in automatico mediante il comando **`npm init`**.
  *Per avere la certezza che tutte le dipendenze vengano gestite correttamente bisognerebbe rimuovere la variabile d'ambiente globale (o più semplicemente rinominare nel profilo utente la cartella **npm** contenente node_modules)*
- Creare nella cartella d lavoro principale il file **`.gitignore`** contenente le seguenti righe (notare la presenza di .gitignore all'interno del file. Non dovrebbe servire ma qualcuno lo mette.

```
#  Node build artifacts
# .gitignore
node_modules
```

- Creare nella cartella d lavoro principale il file **`Procfile`** contenente il nome del file di lancio. Questo file non è indispensabile in quanto, in sua assenza, il file di lancio viene letto da package,json

```
web: node server.js
```

- Eseguire il comando ***git init*** che 'trasforma' la cartella corrente in una "cartella GIT" creando una sottocartella nascosta **`.git`** preposta a contenere l'intero repository e la sua struttura.
- Eseguire il comando ***heroku git:remote -a roberto-mana-app*** che assegna al "server remoto" denominato ***heroku*** la url associata all'applicazione indicata dall'opzione –a
  Questa url viene memorizzata all'interno del file .git / config in modo da poter essere utilizzata al momento dell'upload.

❖ Modifica e upload del progetto

Dopo ogni modifica, occorre eseguire i seguenti passi:

- **git add .**
  Questo comando salva in stage tutti i files aggiornati, compresi tutti gli eventuali **nuovi file** ed un indicatore per tutti i file eventualmente **rimossi**
- **git commit –a -m "beta starting version"**
  Questo comando salva nella cartella .git tutte le snapshot in stage (create dal comando precedente). L'opzione –a garantisce il commit di tutti i file non in stage ma modificati (senza però contemplare quelli nuovi e quelli rimossi). L'opzione **-m** assegna un commento alla nuova versione, commento che viene memorizzato all'interno del file .git / COMMIT_EDITMSG
- **git push heroku master** oppure **git push heroku main** oppure **git push heroku HEAD:master**
  Eseguel'UPLOAD del progetto sul server HEROKU all'interno della URL generata al punto precedente
- In caso di errori durante l'upload utilizzare il comando **`heroku logs --tail`**
  Il comando **`heroku logs -n 100`** visaulizza gli utlimi 100 log

❖ Eventuale Download di una applicazione di esempio

- Da una qualsiasi cartella eseguire il comando
  git clone https://github.com/heroku/node-js-getting-started.git
  Questo comando crea una sottocartella **node-js-getting-started** all'interno della quale scarica un progetto di esempio contenente tutti i files necesari alla pubblicazione.
  Il file **`index.js`** rappresenta l'entry point del progetto
- Entrare nella sottocartella contenente il progetto (***cd*** **node-js-getting-started**)
- Lanciare ***npm install*** per scaricare tutte le dipendenze indicate all'interno del file package.json e necessarie all'esecuzione locale del progetto
- Avviare il server tramite il comando ***node index*** oppure ***npm start*** che avvia sulla porta 5000 il file indicato all'interno di Procfile oppure, in sua assenza, all'interno di package.json
- Aprire il browser e connettersi a localhost:5000. L'applicazione di prova contiene anch'essa tutti i vari passi necessari alla pubblicazione su heroku

❖ Manutenzione del progetto

- Una volta registrati è possibile accedere via web a **heroku.com** per apportare modifiche alla configurazione del nostro progetto e/o per creare nuovi progetti paralleli.
- Nuovi progetti possono essere creati anche da terminale con il comando ***heroku create*** `appName`
  Se non si specifica un appName, heroku assegna al nuovo progetto una URL generata casualmente dal provider. Ad esempio:

  ```
  https://obscure-anchorage-56060.herokuapp.com/
  https://git.heroku.com/obscure-anchorage-56060.git
  ```

- Il comando da terminale ***heroku ps*** consente di visualizzare le ore di utilizzo rimanenti

❖ Altri comandi

- Il comando "**heroku open**" apre la URL del progetto sul proprio browser senza doverla digitare.
- Il comando "**heroku local web**", avvia un server di prova locale che esegue la app remota. Digitando nel browser *localhost:5000* viene visualizzata la pagina di apertura

Connessione a mongoDB Atlas

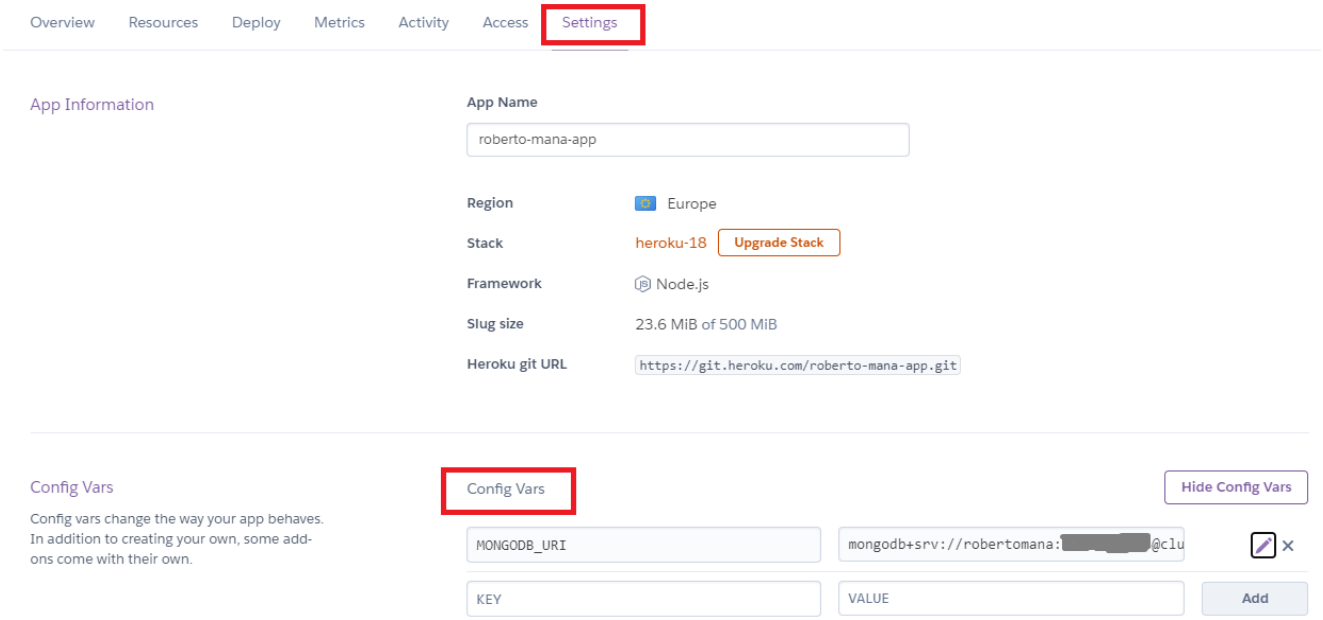
Per connettere la nostra APP ad un database di Atlas occorre istruire heroku riguardo alla modalità di connessione con Atlas (variabile **MONGODB_URI**). Questo può essere fatto in due modi:

da terminale

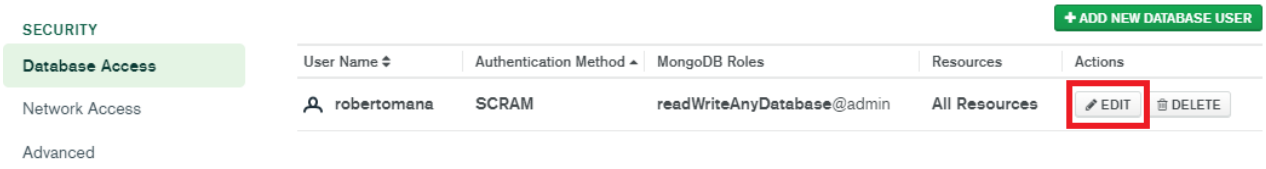
```
heroku config:set MONGODB_URI=“atlas connection string”
// senza spazi prima e dopo del segno di uguale !
```

dall’interfaccia grafica di heroku

sottofinestra settings della app corrente



Attenzione che si cambia la password di accesso su Atlas, e l’utente di accesso è lo stesso utilizzato per accedere al database, tale password deve essere impostata anche sull’utente di accesso al database(finestra **Security / Database Access)**. Molto meglio tenere le due cose distinte e creare un utente specifico per l’accesso al database con la sua password



A questo punto non è più necessario impostare la connectionString all’interno dell’applicazione, dove sarà sufficiente scrivere:

```
  const CONNECTIONSTRING = process.env.MONGODB_URI     // heroku app
// const CONNECTIONSTRING = "atlas connectionString"     // app locale DBMS atlas
// const CONNECTIONSTRING = "mongodb://127.0.0.1:27017"  // app locale DBMS locale
```

Una eventuale OR con `"mongodb://127.0.0.1:27017"` sembra dare problemi in fase di prima esecuzione. Meglio gestire tre connectionString separate da utilizzare nei vari casi
Le varie routes di accesso al database rimangono immutate.

Utilizzo dei web socket

`https://devcenter.heroku.com/articles/node-websockets`

Per poter utilizzare la libreria socket.io su heroku occorre eseguire da terminale la seguente abilitazione:

```
heroku features:enable http-session-affinity
```

File storage su cloudinary.com

Molto spesso i web server, specie nelle versioni free, non consentono lo storage dei file.
Cloudinary è un server di storage gratuito e di facile utilizzo. Viene usato principalmente per le immagini per le quali dispone di diversi strumenti di editing che consentono di applicare alle immagini effetti anche molto interessanti.

Una volta eseguita la registrazione ed il login, i file possono essere semplicemente trascinati nell'apposita area oppure **uploadati** mediante l'utilizzo qualsiasi linguaggio di programmazione. Esistono librerie di accesso a cloudinary in quasi tutti i linguaggi di programmazione.

HTTP | Ruby | PHP v1 | PHP v2 | Python | Node.js | Java | JS | jQuery | React | Vue.js | Angular | .NET | Android | iOS

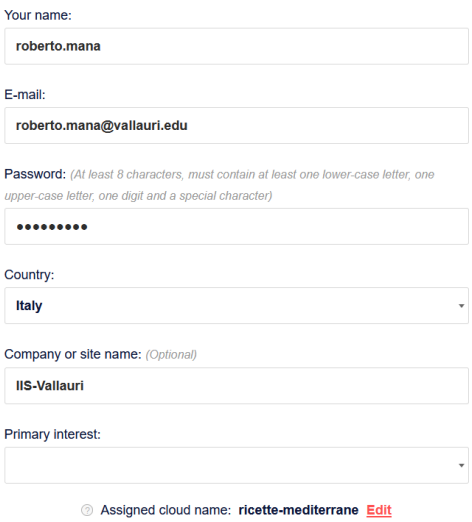
In corrispondenza di un upload, cloudinary restituisce una URL pubblica della posizione in cui è stato salvato il file. Lo storage dei file può essere fatto in due modi:

- Direttamente dal client che poi invia al server la URL ricevuta da cloudinary
- Il client invia il file al server in formato binario oppure in formato base64 ed il server provvederà lui a salvarlo su cloudinary che gli restituirà la URL da salvare nel database.

L'upload fatto dal server è sicuramente più pesante ma anche più sicuro perchè può essere protetto con le chiavi di accesso a cloudinary. Viceversa, nel caso del client, è assolutamente sconsigliato salvare le chiavi sul client che quindi, solitamente, esegue i propri upload senza alcuna autenticazione. Molti esempi della documentazione ufficiale sono disponibili soltanto per i linguaggi lato server.

Registrazione

In fase di registrazione occorre definire un **`cloud name`** che dovrà poi essere utilizzato per l'accesso da codice. Il **`cloud name`** deve soddisfare le solite regole delle URL: sono ammessi caratteri minuscoli, numeri ed il trattino. Ad esempio **roberto-mana** oppure **ricette-mediterranee**



A seguito di una registrazione cloudinary assegna all'utente uno spazio gratuito pari a **1 GB**.
Per dimensioni maggiori ci sono diversi altri profili a pagamento.

L'utente può organizzare questo spazio a suo piacimento creando nuove sottocartelle e trascinando nelle sottocartelle i propri file. All'inizio è presente una unica sottocartella denominata **samples** contenente alcune immagini di esempio.

Al termine della registrazione vengono visualizzate alcune finestre che riportano, ad esempio, le credenziali di accesso da codice, credenziali che saranno comunque sempre visibili nella dashboard. Viene visualizzato anche un esempio di utilizzo nei vari ambienti.

Utilizzo lato server tramite nodejs

Il client deve inviare l'immagine al web server come parametro in **formato base64** oppure in **formato binario** e poi provvederà il server ad uploadare l'immagine su cloudinary e a salvare il path pubblico all'interno del database.

https://cloudinary.com/visualweb/display/IMMC/Node.js+Image+Upload
https://cloudinary.com/documentation/node_integration

Codice suggerito da cloudinary:

upload code

```
cloudinary.uploader.upload("sample.jpg",
    {"crop":"limit",  "tags":"samples",  "width":3000,  "height":2000},
    function(result) {
              console.log(result)
});
```

image manipulation tag

```
cloudinary.image("sample",
{"crop":"fill","gravity":"faces","width":300,"height":200,"format":"jpg"});
```

Configurazione

```
const CLOUD_NAME = "ricette-mediterranee"
const API_KEY = "673959956397347"
const API_SECRET = "********************"
const CLOUDINARY_URL =
 "cloudinary://673959956397347:*****************@ricette-mediterranee"
```

la `CLOUDINARY_URL` è l'unione delle tre precedenti

```
const cloudinary = require('cloudinary').v2
cloudinary.config({
      cloud_name: CLOUD_NAME,
      api_key: API_KEY,
      api_secret: API_SECRET,
      // secure:true    // https
});
```

Upload di una immagine base64

Sia **`req.body.image`** l'immagine base64 inviata dal client al server nodejs.
Per eseguire l'upload su cloudinary si può eseguire il seguente codice:

```
app.post("/api/uploadImage/", function(req, res, next){
    cloudinary.uploader.upload(req.body.image)
    .then((result) => {
        res.send({"url":result.secure_url})
     })
    .catch((error) => {
        res.status(500).send("error uploading file")
     })
})
```

dove **result** è un json contenente, tra le altre cose, all'interno del campo **secure_url** la **url** pubblica assegnata da cloudinary all'immagine. Nelle caso delle immagini **base64** le immagini vengono trasmesse senza un filename, per cui Cloudinary ne assegna uno proprio che non sembra in nessun modo modificabile (nemmeno dalla dashboard)

```
{
  asset_id: 'b8246a3779ed2c7889aaf1118219ce25',
  public_id: 'fhneqkvepqey9iceyzav',
  version: 1617889660,
  version_id: '216ef74442fa57cf3deec888b403f927',
  signature: 'b58250fd6dbc2023c56f974c028fd7fa0770f1d2',
  width: 300,
  height: 225,
  format: 'jpg',
  resource_type: 'image',
  created_at: '2021-04-08T13:47:40Z',
  tags: [],
  bytes: 42165,
  type: 'upload',
  etag: '5f00140464423022c6be23e2a55a28bf',
  placeholder: false,
  url: 'http://res.cloudinary.com/
       ricette-mediterranee/image/upload/v1617889660/fhneqkvepqey9iceyzav.jpg',
  secure_url: 'https://res.cloudinary.com/
       ricette-mediterranee/image/upload/v1617889660/fhneqkvepqey9iceyzav.jpg'
}
```

Indicazione della cartella dove salvare il file

E' possibile passare al metodo **upload()** come secondo parametro un json di opzioni in cui si può specificare, ad esempio, la cartella del proprio spazio su cloudinary in cui salvare il file.
Se la cartella non esiste viene automaticamente creata.
L'opzione **use_filename:true** chiede a cloudinary di mantenere il nome del file. Però, come detto, le immagini base64 non hanno nome, per cui l'opzione non ha nessun effetto.

```
cloudinary.uploader.upload(req.body.image,{folder:"ricette", use_filename:true},
    (err, result) => {
         collection.updateOne(
           {"_id":ObjectId(id)}, {"$set":{"image":result.secure_url}},
         (err, data) => {
            if (err)
               res.status(500).send("Internal Error");
            else
               res.json({ "result": "ok" });
            client.close();
         });
    });
```

Upoad di una immagine binaria

Al metodo **cloudinary.uploader.upload** si può passare indifferentemente:

- Una immagine in formato base64
- Il path di una immagine su disco

Sia **path** il path completo dell'immagine salvata provvisoriamente su server

```
let folderPath = path.split('/');
folderPath.pop();
folderPath = folderPath.join('/');
if (!fs.existsSync("./static" + folderPath))
     fs.mkdirSync("./static" + folderPath);

file.mv("./static" + path, (err, data) => {
     if (err) {
          log("Errore query: " + err.errmsg);
          res.json({ "error": "Errore query: " + err.errmsg });
     }
     else {
          cloudinary.uploader.upload("./static" + path,
            {folder:id, use_filename:true},
           (err, result) => { }
     }
```

Il metodo cloudinary.image(filename)

E' un metodo **sincrono** che, partendo dal nome di una immagine (nome che può essere letto sulla dashboard oppure dedotto dalla URL mediante uno split) restituisce il tag completo di quella immagine comprensivo della url assoluto, tag che può così essere aggiunto dinamicamente alla pagina.

```
let ris=cloudinary.image("miofile.jpg")
// valore restituito :
<img src='http://res.cloudinary.com/ricette-
mediterranee/image/upload/miofile.jpg'/>
```

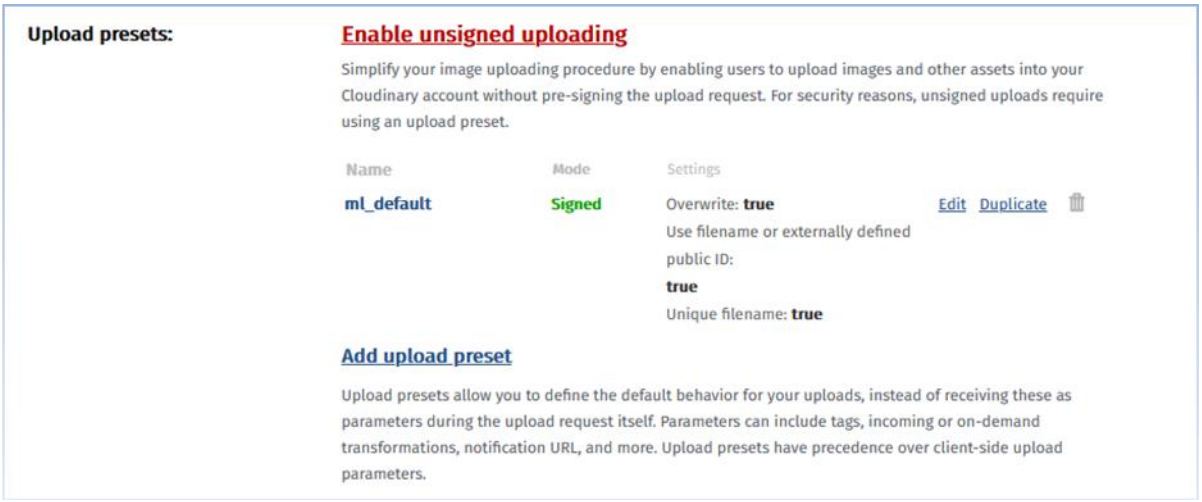
Se si dispone della url pubblica, questo metodo è abbastanza inutile. Va bene quando non si dispone della url pubblica ma si dispone soltanto del nome del file, perché è stato uploadato ad esempio dal client con l'opzione **use_filename:true** leggendo il file direttamente da disco.

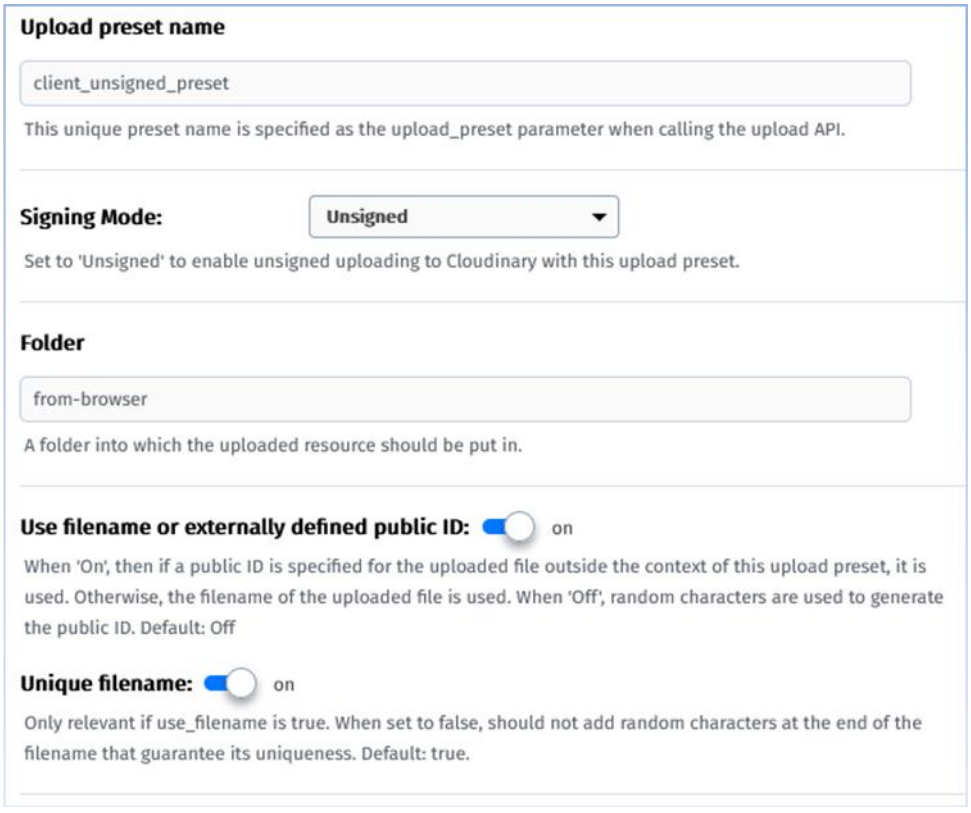
Utilizzo lato client tramite jquery

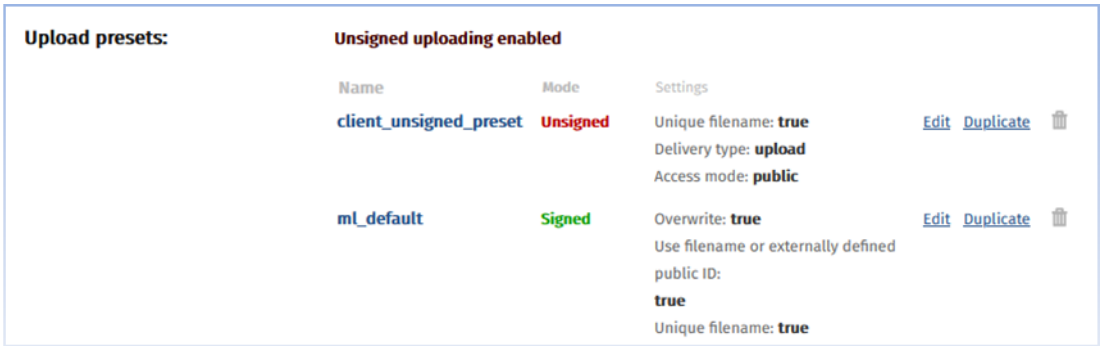
Per poter eseguire l'upload di immagini lato client occorre, su cloudinary, andare su settings/upload e:

- abilitare **l'unsigned uploading**
- creare un **upload preset** di tipo unsigned da utilizzare negli upload da client

Come si può vedere dalla figura esiste giù un **upload preset** di default, di tipo signed con nome **ml_default**


Upload presets:
Enable unsigned uploading
Simplify your image uploading procedure by enabling users to upload images and other assets into your Cloudinary account without pre-signing the upload request. For security reasons, unsigned uploads require using an upload preset.
Name
Mode
Settings
ml_default
Signed
Overwrite: true
Use filename or externally defined public ID:
true
Unique filename: true
Edit
Duplicate
Add upload preset
Upload presets allow you to define the default behavior for your uploads, instead of receiving these as parameters during the upload request itself. Parameters can include tags, incoming or on-demand transformations, notification URL, and more. Upload presets have precedence over client-side upload parameters.


Upload preset name
client_unsigned_preset
This unique preset name is specified as the upload_preset parameter when calling the upload API.
Signing Mode:
Unsigned
Set to 'Unsigned' to enable unsigned uploading to Cloudinary with this upload preset.
Folder
from-browser
A folder into which the uploaded resource should be put in.
Use filename or externally defined public ID:
on
When 'On', then if a public ID is specified for the uploaded file outside the context of this upload preset, it is used. Otherwise, the filename of the uploaded file is used. When 'Off', random characters are used to generate the public ID. Default: Off
Unique filename:
on
Only relevant if use_filename is true. When set to false, should not add random characters at the end of the filename that guarantee its uniqueness. Default: true.


Upload presets:
Unsigned uploading enabled
Name
Mode
Settings
client_unsigned_preset
Unsigned
Unique filename: true
Delivery type: upload
Access mode: public
Edit
Duplicate
ml_default
Signed
Overwrite: true
Use filename or externally defined public ID:
true
Unique filename: true
Edit
Duplicate

E' possibile creare un upload preset anche da codice, utilizzando ad esempio nodejs:

```
const cloudinary = require('cloudinary').v2
cloudinary.api.create_upload_preset(
 {
    cloud_name: CLOUD_NAME,
    api_key: API_KEY,
    api_secret: API_SECRET,
    name: "client_unsigned_preset",
    unsigned: true,
    folder: "new-products",
    categorization: "google_tagging,google_video_tagging",
    auto_tagging: 0.75,
    background_removal: "cloudinary_ai"
  },
  function(error, result){console.log(result)}
);
```

L'opzione **folder** specifica la sotto-cartella in cui salvare l'immagine
L'opzione **Use filename or externally defined public ID** indica di mantenere il nome originale del file. L'opzione successiva (**unique filename**) (selezionata automaticamente insieme alla precedente) indica che il filename deve essere univoco. In caso di filename già esistente vengono concatenati di caratteri casuali in coda.

HTML

```
<!--
      npm install jquery
      npm install blueimp-file-upload
      npm install cloudinary-jquery-file-upload
-->

<html>
<head lang="en">
    <meta charset="UTF-8">
      <title>Upload</title>
      <script  src="jquery/jquery.min.js"></script>
      <script src="blueimp/js/vendor/jquery.ui.widget.js"></script>
      <script src="blueimp/js/jquery.iframe-transport.js"></script>
      <script src="blueimp/js/jquery.fileupload.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/cloudinary-jquery-
               file-upload/2.11.3/cloudinary-jquery-file-
upload.min.js"></script>
      <script src="index.js"> </script>
</head>

<body>
      <h1>Upload di un file</h1>
      <form>
            <h4>Select a file</h4>
            <input type="file" name="file" id="txtFile" >
      </form>
      <div class="gallery" style="width:200px; margin:20px;"/>
</body>
</html>
```

JS

```
$(document).ready(function() {
      const CLOUD_NAME = "ricette-mediterranee"
      const unsignedUploadPreset = 'client_unsigned_preset';

    $.cloudinary.config({
       cloud_name: CLOUD_NAME,
       // secure: true
    });

    let options = {
       cloudName: CLOUD_NAME,
       tags: 'browser_uploads'
    }

    // initializing the input field for use with the cloudinary_fileupload
method:
    $('#txtFile').unsigned_cloudinary_upload(unsignedUploadPreset,
                                             options, {multiple: true})
    .bind('cloudinarydone', function(e, data) {
       console.log('Upload result:')
       console.log(data.result);
       // Create a thumbnail of the uploaded image, with 150px width
       var image = $.cloudinary.image(
           data.result.public_id, {secure:true, width:150, crop:'scale'})
       $('.gallery').prepend(image);
    });
})
```