

BrainJobs

UniShare

Davide Cozzi
@dlcgold

Gabriele De Rosa
@derogab

Indice

| | | |
|---|-------------------------|----|
| 1 | Introduzione | 2 |
| 2 | FrontEnd | 4 |
| 3 | Backend | 22 |
| 4 | Gateway | 30 |
| 5 | Esecuzione del Progetto | 35 |
| 6 | Docker | 37 |

Capitolo 1

Introduzione

BrainJobs è un (ipotetico) servizio cloud di tipo Software-as-a-Service (SaaS) che offre ai suoi utenti la possibilità di “allenare” modelli di apprendimento automatico, di valutarne le prestazioni ed (eventualmente) riutilizzarli per effettuare simulazioni. Il sistema permette agli utenti di effettuare richieste di allenamento o simulazione caricando i dati insieme al modello o utilizzando uno già precedentemente allenato e salvato nel proprio archivio. In base al linguaggio o al framework utilizzato per il codice del modello, BrainJobs lancia la computazione in un particolare ambiente di esecuzione che verrà istanziato “on-the-fly” in un’altra piattaforma cloud di tipo Serverless basata su containers (es: Apache OpenWhisk, Knative, ...). Gli utenti possono sottoporre più richieste consecutive. Esse verranno gestite in parallelo in un sistema a coda. Ogni richiesta di un utente corrisponde ad un task di lavoro (job). Gli utenti possono controllare lo stato delle loro richieste dalla dashboard di BrainJobs, ed una volta terminate, visualizzarne i risultati. Successivamente, il sistema permette di scartare o salvare il modello per utilizzi futuri. L’architettura del servizio BrainJobs è suddivisa in tanti servizi e componenti, ognuno con un compito ben specifico. Al vostro team, è richiesta la creazione di due componenti:

1. un componente di frontend implementato utilizzando HTML, CSS e JavaScript che utilizza il paradigma AJAX per inviare/ricevere dati
2. un componente di backend che espone una HTTP API REST

Il frontend deve permettere ad un utente di creare una nuova richiesta di allenamento, visualizzare la lista delle sue richieste e visualizzare le informazioni di dettaglio di ogni richiesta. Il backend deve essere in grado di salvare una nuova richiesta, fornire la lista delle richieste di un utente e restituire informazioni di dettaglio di ogni richiesta. Una volta che il backend ha salvato

una nuova richiesta, altri servizi di BrainJobs si occuperanno di lanciare la computazione, aggiornare lo stato del job ed aggiungere i risultati.

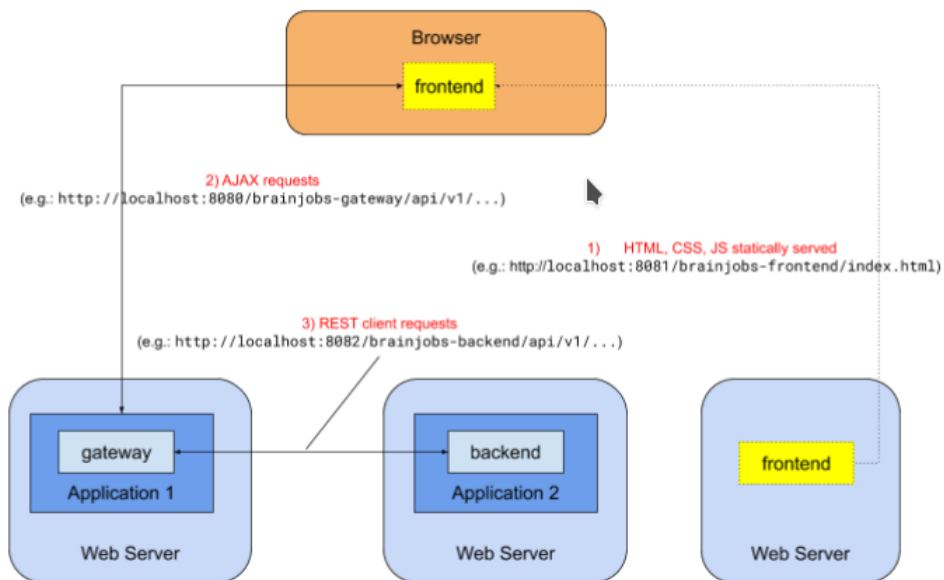


Figura 1.1: struttura finale del progetto

Capitolo 2

FrontEnd

Iniziamo parlando del frontend. Per quanto riguarda l'aspetto estetico è stato usato un tema di Bootstrap per poter rappresentare più semplicemente componenti come il menù presente nella parte alta della pagina, contenente le informazioni del progetto, e la navbar con il bottone per richiamare il menù. Entrambi componenti sono nel tag *header*.



Figura 2.1: navbar

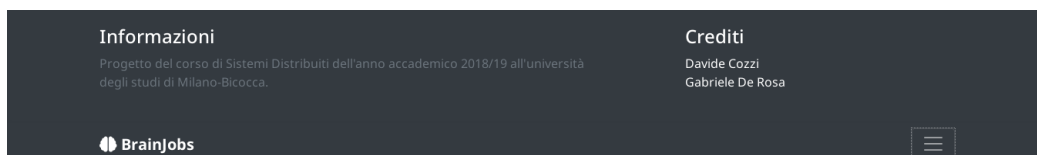


Figura 2.2: menù a scomparsa

navbar:

```
<div class="navbar navbar-dark bg-dark shadow-sm">
  <div class="container d-flex justify-content-between">
    <a href="#" class="navbar-brand d-flex align-items-center">
      <strong><i class="fas fa-brain"></i> BrainJobs</strong>
    </a>
```

```
<button class="navbar-toggler" type="button"
  data-toggle="collapse"
  data-target="#navbarHeader" aria-controls="navbarHeader"
  aria-expanded="false" aria-label="Toggle navigation">
  <span class="navbar-toggler-icon"></span>
</button>
</div>
</div>
```

Menù:

```
<div class="collapse bg-dark" id="navbarHeader">
  <div class="container">
    <div class="row">
      <div class="col-sm-8 col-md-7 py-4">
        <h4 class="text-white">Informazioni</h4>
        <p class="text-muted">
          Progetto del corso di Sistemi Distribuiti dell'anno
          accademico 2018/19 all'università degli studi di
          Milano-Bicocca.
        </p>
      </div>
      <div class="col-sm-4 offset-md-1 py-4">
        <h4 class="text-white">Crediti</h4>
        <ul class="list-unstyled">
          <li><a class="text-white"
            href="https://www.github.com/dlccgold">
              Davide Cozzi</a></li>
          <li><a class="text-white"
            href="https://www.github.com/derogab">
              Gabriele De Rosa</a></li>
        </ul>
      </div>
    </div>
  </div>
</div>
```

Si passa poi alla sezione con il titolo della pagina, con l'icona presa da quella messe a disposizione sul sito <https://fontawesome.com/icons/>:



Figura 2.3: titolo della pagina

```
<section class="jumbotron text-center">
  <div class="container">
    <h1 class="jumbotron-heading"><i class="fas fa-brain"></i>
      BrainJobs</h1>
    </div>
  </section>
```

Analizziamo ora una delle parti principali della pagina: **il form di inserimento dati**. Qui si è fatto uso della classe *form-group* per impostare i vari campi del form, con titolo e *form-control* per l'inserimento, e della classe *custom-select* per quei campi con selezione obbligatoria, dove quindi è stato aggiunto un selettore:

Richiesta di allenamento

| | | |
|-----------|---|-------------------------------|
| user_id | <input type="text" value="Inserisci user_id (obbligatorio)"/> | |
| title | <input type="text" value="Inserisci title (obbligatorio)"/> | |
| language | <input type="text" value=""/> | |
| framework | <input type="text" value=""/> | |
| dataset | <input type="text" value="Inserisci dataset (obbligatorio)"/> | <input type="text" value=""/> |
| model | <input type="text" value="Inserisci model (obbligatorio)"/> | |

Figura 2.4: form per l'inserimento della richiesta

Richiesta di allenamento

user_id

title

language

framework

dataset

model

Pytorch

Tensorflow

Caffe

Keras

Deeplearning4j

Apache_mahout

Apache_singa

© BrainJobs

Figura 2.5: esempio di selettore

Vediamo quindi, per esempio, la parte nell'*index.html* dedicata all'inserimento di *title*, ovvero un inserimento manuale senza selettore:

```
<div class="row">
  <div class="col-md-2 col-sm-12">
    <label for="title">title</label>
  </div>
  <div class="col-md-10 col-sm-12">
    <input type="text" class="form-control" id="title"
      placeholder="Inserisci title (obbligatorio)">
  </div>
</div>
```

Dove notiamo come le classi *col-md-n* e *col-sm-n* permettono di rendere *responsive* gli elementi (la stringa e il box di inserimento).

Passiamo ora a vedere l'esempio di un inserimento mediante selettore, prendendo come esempio l'inserimento del framework:

```
<div class="form-group">
  <div class="row">
    <div class="col-md-2 col-sm-12">
      <label for="framework">framework</label>
    </div>
    <div class="col-md-10 col-sm-12">
      <select class="custom-select" class="form-control"
        id="framework">
        <option selected></option>
        <option value="pytorch">Pytorch</option>
        <option value="tensorflow">Tensorflow</option>
        <option value="caffe">Caffe</option>
        <option value="keras">Keras</option>
        <option value="deeplearning4j">Deeplearning4j</option>
        <option value="apache_mahout">Apache_mahout</option>
        <option value="apache_singa">Apache_singa</option>
      </select>
    </div>
  </div>
</div>
```

Analizziamo ora la seconda parte fondamentale della pagina, dove l'utente può inserire uno *user_id* o un *job_id* per effettuare una query nel database delle richieste:

Dettagli

Richieste di uno user

Richiesta singola

Figura 2.6: campi per l'inserimento di query

Nell'*index.html* si ha quindi:

```
<h3>Dettagli</h3>

<h6>Richieste di uno user</h6>
<div class="row">
  <div class="col-md-6">
    <input type="text" class="form-control"
      id="user_id_search"
      placeholder="Inserisci lo user_id">
  </div>
  <div class="col-md-6">
    <button id="get-all-requests"
      class="btn btn-dark btn-block">
      Visualizza le richieste di user</button>
  </div>
</div>

<h6 style="margin-top: 10px;">Richiesta singola</h6>
<div class="row">
  <div class="col-md-6">
    <input type="text" class="form-control"
      id="job_id" placeholder="Inserisci job_id">
  </div>
  <div class="col-md-6">
    <button id="get-single-request"
      class="btn btn-dark btn-block">
      Visualizza la richiesta</button>
  </div>
</div>
```

infine i risultati della query verranno visualizzati mediante:

```
<div id="results"></div>
```

Infine una parola per tutta quella parte del file dedicata al permettere l'uso di *bootstrap*, *jquery* e del *custom.js* mediante il quale, con l'uso di **ajax**, sono state fatte le *POST* e le *GET*:

```
<!-- nell'HEAD -->

<!-- Bootstrap core CSS -->
<link href="css/bootstrap.min.css" rel="stylesheet">

<!-- FA -->
<link href="css/fa.css" rel="stylesheet">

<!-- Custom styles for this template -->
<link href="css/custom.css" rel="stylesheet">

...

<!-- alla fine del file -->

<!-- jQuery -->
<script src="js/jquery.min.js"></script>
<script>window.jQuery ||
  document.write('<script src="js/jquery.min.js">
    </script>')</script>

<!-- Bootstrap bundle JS -->
<script src="js/bootstrap.bundle.min.js"></script>

<!-- Custom javascript script w/ ajax requests-->
<script src="js/custom.js"></script>
```

Come footer si ha l'icona del copyright e un “torna su” che è collegato ad un'ancora all'inizio dell'html:

```
<footer class="text-muted">
  <div class="container">
```

```
<p class="float-right">
  <a href="#">Torna su</a>
</p>
<p><i class="far fa-copyright"></i> BrainJobs 2019</p>
</div>
</footer>
```

Abbiamo visto il file *HTML* ma questo non basta in quanto serve il *custom.js* per interagire, mediante jquery e ajax, con il backend. Si ha quindi la seguente struttura per il **frontend**:

```
.
  css:
    bootstrap.css
    bootstrap.css.map
    bootstrap-grid.css
    bootstrap-grid.css.map
    bootstrap-grid.min.css
    bootstrap-grid.min.css.map
    bootstrap.min.css
    bootstrap.min.css.map
    bootstrap-reboot.css
    bootstrap-reboot.css.map
    bootstrap-reboot.min.css
    bootstrap-reboot.min.css.map
    custom.css
    fa.css
    fa.min.css
  index.html
  js:
    bootstrap.bundle.js
    bootstrap.bundle.js.map
    bootstrap.bundle.min.js
    bootstrap.bundle.min.js.map
    bootstrap.js
    bootstrap.js.map
    bootstrap.min.js
    bootstrap.min.js.map
```

```
custom.js
fa.js
fa.min.js
jquery.js
jquery.min.js
webfonts:
  fa-brands-400.eot
  fa-brands-400.svg
  fa-brands-400.ttf
  fa-brands-400.woff
  fa-brands-400.woff2
  fa-regular-400.eot
  fa-regular-400.svg
  fa-regular-400.ttf
  fa-regular-400.woff
  fa-regular-400.woff2
  fa-solid-900.eot
  fa-solid-900.svg
  fa-solid-900.ttf
  fa-solid-900.woff
  fa-solid-900.woff2
```

con tutti i file per la parte di CSS, tutto il necessario per jquery e bootstrap, i fonts etc...

Ci concentriamo ovviamente sul *custom.js* che abbiamo scritto per interfacciare frontend e backend.

Aperto il file notiamo innanzitutto una definizione di costante:

```
const API = "http://localhost:8080/brainjobs-gateway/api/v1/";
```

questa rappresenta parte del path per i vari endpoint presenti nel gateway. Proseguendo oltre incontriamo la prima funzione di **jQuery** che controlla che quanto è contenuto al suo interno venga eseguito unicamente una volta che il **DOM** sia pronto per l'esecuzione di codice javascript:

```
$(document).ready(function(){  
  ...  
});
```

Passiamo ora ad analizzare le tre operazioni base che vengono effettuate. Nell'*index.html* il bottone associato all'invio di una richiesta era stato identificato con l'id *send-form* possiamo quindi sfruttare il metodo *.click()* di jquery per triggerare una funzione *function(e)* con una sequenza di istruzioni ogni volta che viene clickato il bottone. Vediamo quindi queste istruzioni nel caso dell'invio del form.

Innanzitutto abbiamo un'istruzione per prevenire l'azione di default di un evento (*e*), evitando quindi il refresh della pagina:

```
e.preventDefault();
```

Definiamo poi una variabile per ogni campo del form sfruttando il metodo di jquery *.val()* che restituisce il valore di un id, con l'aggiunta del *.trim()* che rimuove eventuali spazi all'inizio e alla fine:

```
var user_id = $("#user_id").val().trim();  
var title = $("#title").val().trim();  
var language = $("#language").val().trim();  
var framework = $("#framework").val().trim();  
var dataset = $("#dataset").val().trim();  
var dataset_datatype = $("#dataset_datatype").val().trim();  
var model = $("#model").val().trim();
```

La traccia impone l'obbligo di avere tutti i campi compilati tranne quello riguardante il framework, che è opzionale. Una if con la condizione che basti l'assenza di uno dei campi viene usata per restituire, mediante l'id *send-form-result* e il metodo *.html()*, un alert, con un bottone dedicato alla sua chiusura:

```
if (!user_id || !title || !language ||
    !dataset || !dataset_datatype || !model ) {
$('#send-form-result').html(
  '<div class="alert alert-warning alert-dismissible fade show"
  role="alert">'
  + '<strong>Warning!</strong> Compilare i campi obbligatori.'
  + '<button type="button" class="close"
    data-dismiss="alert" aria-label="Close">'
  + '<span aria-hidden="true">&times;</span>'
  + '</button>'
  + '</div>');
}
```

Richiesta di allenamento

Warning! Compilare i campi obbligatori. ×

user_id

Inserisci user_id (obbligatorio)

title

Inserisci title (obbligatorio)

language

framework

dataset

Inserisci dataset (obbligatorio)

model

Inserisci model (obbligatorio)

Figura 2.7: alert nel caso di form incompleto

Nel caso invece di form inserito correttamente si può procedere, mediante l'else, con la richiesta **ajax**.

In informatica AJAX, acronimo di Asynchronous JavaScript and XML, è una tecnica di sviluppo software per la realizzazione di applicazioni web interattive (Rich Internet Application). Lo sviluppo di applicazioni HTML con AJAX si basa su uno scambio di dati in background fra web browser e server, che consente l'aggiornamento dinamico di una pagina web senza esplicito ricaricamento da parte dell'utente.

AJAX è asincrono nel senso che i dati extra sono richiesti al server e caricati in background senza interferire con il comportamento della pagina esistente. Normalmente le funzioni richiamate sono scritte con il linguaggio JavaScript. Tuttavia, e a dispetto del nome, l'uso di JavaScript e di XML non è obbligatorio, come non è detto che le richieste di caricamento debbano essere necessariamente asincrone.

Si usa quindi il metodo `.ajax()`, *acui viene specificato, mediante il campo `type`, il tipo di richiesta, e nel nostro caso queste ultime due funzioni sono due alert che avvisano se la richiesta è stata effettivamente*

Passiamo ora alla prima delle due query, quella nel quale si inserisce uno `user_id` per ottenere tutte le richieste da lui effettuate. Come prima procediamo al click sul bottone, stavolta con l'id `get-all-requests`, e salviamo in una variabile lo `user_id`:

```
$(document).ready(function() {  
    var user_id = $("#user_id_search").val().trim();  
    e.preventDefault();  
    ...  
});
```

Anche qui con un `if` controlliamo che sia stato effettivamente inserito uno `user_id` prima di effettuare la query premendo il bottone e in caso contrario si provvede a stampare un alert dedicato:

```
if( !user_id ) {  
    $('#results').html('<div class="alert alert-warning  
        alert-dismissible fade show" role="alert">')
```



```
+ '<strong>Warning!</strong> Compila il campo user_id.'  
+ '<button type="button" class="close" data-dismiss="alert"  
  " aria-label="Close">'  
+ '<span aria-hidden="true">&times;</span>'  
+ '</button>'  
+ '</div>');  
}
```

Dettagli

Richieste di uno user

Visualizza le richieste di user

Richiesta singola

Visualizza la richiesta

Warning! Compila il campo user_id.



Figura 2.10: alert nel caso di assenza di inserimento dello user

Altrimenti, mediante l'else, effettuiamo una richiesta di tipo *GET* mediante ajax. Si hanno due casi di successo, il primo se la richiesta avviene correttamente ma non si hanno richieste per quello user_id, la seconda se si ha invece almeno una richiesta. Nel primo caso si controlla che l'array contenente le informazioni abbia lunghezza 0, in tal caso si stampa un alert.

Dettagli

Richieste di uno user

Visualizza le richieste di user

Richiesta singola

Visualizza la richiesta

Warning! Nessuna richiesta per lo user selezionato.



Figura 2.11: alert nel caso di assenza di richieste

Nel secondo caso, mediante il metodo `.each(function)` che permette di iterare su un oggetto jquery permettendo di eseguire una *function* per ogni elemento, stampo i vari campi di ogni richiesta che mi viene resitutita dal server salvando progressivamente il tutto in una stringa *result* che verrà associata all'id omonimo nell'*index.html*.

The screenshot shows a web interface titled 'Dettagli'. It has two input sections: 'Richieste di uno user' with a text box containing 'test' and a button 'Visualizza le richieste di user'; and 'Richiesta singola' with a text box 'Inserisci job_id' and a button 'Visualizza la richiesta'. Below these is a light blue box titled 'Dati richiesti' containing the following text:

```
Richiesta 1
user_id: test
title: test
language: python
framework: pytorch
dataset: test
dataset_datatype: json
model: test
status: created
created_at: 01/07/2019
job_id: 3ff24c16-868d-489f-90d3-287a5d5fcfc0

Richiesta 2
user_id: test
```

Figura 2.12: risultati in caso di presenza di richieste

In caso di *error* invece si avrà il solito alert:

The screenshot shows the same 'Dettagli' interface as Figure 2.12, but with an error alert at the bottom. The 'Richieste di uno user' text box now contains 'test4'. The 'Richiesta singola' text box is empty. A red alert box at the bottom contains the text: 'Errore! Si è verificato un errore durante l'invio della richiesta.' with a close button (X) on the right.

Figura 2.13: alert in caso di errore nella richiesta ajax

Per quanto riguarda la seconda query funziona esattamente nella stessa maniera con l'unica differenza che in caso di presenza di una richiesta con

il `job_id` indicato non si ha alcun `.each()` in quanto si può avere una sola richiesta con quel `job_id`:

```
$.ajax({

  type: "GET",
  url: API + 'jobs/' + job_id,
  dataType: "json",
  success: function(data) {

    if(!data){
      $('#results').html('<div class="alert alert-warning  
alert-dismissible fade show" role="alert">'
        + '<strong>Warning!</strong> Questo job_id non esiste.'  
        + '<button type="button" class="close" data-dismiss="alert" data-  
        aria-label="Close">'
        + '<span aria-hidden="true">&times;</span>'
        + '</button>'
        + '</div>');
    }
    else{

      var result = "";
      result += "user_id: " + data.user_id + " <br> ";
      result += "title: " + data.title + " <br> ";
      result += "language: " + data.language + " <br> ";
      result += "framework: " + data.framework + " <br> ";
      result += "dataset: " + data.dataset + " <br> ";
      result += "dataset_datatype: " +  
        data.dataset_datatype + " <br> ";
      result += "model: " + data.model + " <br> ";
      result += "status: " + data.status + " <br> ";
      result += "created_at: " + data.created_at + " <br> ";
      result += "job_id: " + data.job_id + " <br> ";
      result += "<br>";

      $('#results').html('<div class="alert alert-info"  
        role="alert">'
        + '<strong>Dati richiesti</strong> <br><br>' + result
```

```
    + '</div>');  
  
  }  
  
},  
error: function() {  
  $('#results').html('<div class="alert alert-danger  
    alert-dismissible fade show" role="alert">  
+ '<strong>Errore!</strong> Si è verificato un  
    errore durante l\'invio della richiesta.'  
+ '<button type="button" class="close" data-dismiss="alert"  
    aria-label="Close">  
    + '<span aria-hidden="true">&times;</span>'  
+ '</button>'  
+ '</div>');  
}  
  
});
```

Si ha quindi:

Dettagli

Richieste di uno user

Visualizza le richieste di user

Richiesta singola

Visualizza la richiesta

Dati richiesti
user_id: test
title: test
language: python
framework: pytorch
dataset: test
dataset_datatype: json
model: test
status: created
created_at: 01/07/2019
job_id: 3ff24c16-868d-489f-90d3-287a5d5fcfc0

Figura 2.14: risultato in caso di successo

Dettagli

Richieste di uno user

Visualizza le richieste di user

Richiesta singola

Visualizza la richiesta

Warning! Questo job_id non esiste. ✕

Figura 2.15: alert mancata presenza di richieste per quel job_id

Dettagli

Richieste di uno user

Visualizza le richieste di user

Richiesta singola

Visualizza la richiesta

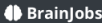
Errore! Si è verificato un errore durante l'invio della richiesta. ✕


Figura 2.16: alert nel caso di errore della richiesta

Quindi nel complesso la pagina si presenta così:

Informazioni
Progetto del corso di Sistemi Distribuiti dell'anno accademico 2018/19 all'università degli studi di Milano-Bicocca.

Crediti
Davide Cozzi
Gabriele De Rosa

 BrainJobs



Richiesta di allenamento

Richiesta ricevuta! ×

user_id

test2

title

test2

language

java

framework

Deeplearning4j

dataset

test

json

model

test

Invia la richiesta

Dettagli

Richieste di uno user

test2

Visualizza le richieste di user

Richiesta singola

3ff24c16-868d-489f-90d3-

Visualizza la richiesta

Dati richiesti

Richiesta 1

user_id: test2

title: test2

language: java

framework: deeplearning4j

dataset: test

dataset_datatype: json

model: test

status: created

created_at: 01/07/2019

job_id: 60ae00d3-1572-4497-ab21-f7c8ed770313

© BrainJobs 2019

Torna su

Figura 2.17: frontend brainjob

Capitolo 3

Backend

Per il backend ci siamo appoggiati su **nodejs**:

Node.js è una runtime di JavaScript Open source multiplatforma orientata agli eventi per l'esecuzione di codice JavaScript Server-side, costruita sul motore JavaScript V8 di Google Chrome. Molti dei suoi moduli base sono scritti in JavaScript, e gli sviluppatori possono scrivere nuovi moduli in JavaScript.

In origine JavaScript veniva utilizzato principalmente lato client. In questo scenario gli script JavaScript, generalmente incorporati all'interno dell'HTML di una pagina web, vengono interpretati da un motore di esecuzione incorporato direttamente all'interno di un Browser. Node.js consente invece di utilizzare JavaScript anche per scrivere codice da eseguire lato server, ad esempio per la produzione del contenuto delle pagine web dinamiche prima che la pagina venga inviata al Browser dell'utente. Node.js in questo modo permette di implementare il cosiddetto paradigma "JavaScript everywhere" (JavaScript ovunque), unificando lo sviluppo di applicazioni Web intorno ad un unico linguaggio di programmazione (JavaScript).

Node.js ha un'architettura orientata agli eventi che rende possibile l'I/O asincrono. Questo design punta ad ottimizzare il Throughput e la scalabilità nelle applicazioni web con molte operazioni di input/output, è inoltre ottimo per applicazioni web Real-time (ad esempio programmi di comunicazione in tempo reale o browser game).

Node.js è un progetto di sviluppo Open source distribuito gestito dalla Node.js Foundation e facilitato tramite il programma di progetti collaborativi della Linux Foundation.

La struttura del backend è la seguente:

```
brainjobs-backend
  api
    index.js
  db.json
  index.js
  node_modules
  package.json
  package-lock.json
```

Partiamo con l'analizzare l'*index.js* presente subito nella cartella *brainjobs-backend*. Qui vengono definiti tutti i requisiti necessari per l'uso delle porte (8082 per il backend e 8081 per il frontend). Come database d'appoggio è stato scelto *lowdb*, <https://github.com/typicode/lowdb>, ovvero un database minimale basato su file json che consente un facile utilizzo sia per quanto riguarda l'inizializzazione dello stesso che per quanto riguarda le query. All'interno del file viene quindi inizializzato il file *db.json* e ne vengono settati i valori di default:

```
const low = require('lowdb');
const FileSync = require('lowdb/adapters/FileSync');

const adapter = new FileSync('db.json');
const db = low(adapter);

db.defaults({ requests: [] }).write();
```

È stato scelto di usare un framework per web application chiamato *express*, <https://github.com/expressjs/express>. Usando *express* è facile istanziare una web app (usando il metodo *.use()*) con metodi GET e POST. Sempre con *express* è possibile gestire il frontend in maniera statica, mediante *express.static()*, (mentre il backend verrà eseguito dinamicamente). Con *require* si richiama l'*index.js* contenuto nella sottocartella */api* al quale viene passato come argomento la web app generata da *express* e il database.


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

// backend
var app_backend = express();
var port_backend = 8082;
// frontend
var app_frontend = express();
var port_frontend = 8081;

console.log('Web Server started. ');
console.log('API started on: ' + port_backend);
console.log('Static index on: ' + port_frontend);

// API
require(__dirname + '/api')(app_backend, db);

// FRONT-END
app_frontend.use('/brainjobs-frontend',
  express.static('../brainjobs-frontend/'));
```

Viene anche usato *body-parse*, <https://github.com/expressjs/body-parser>, che parse il body della richiesta in un middleware, raggiungibile mediante *req.body*, usando il metodo *.urlencoded(options)* per il parsing di dati nel formato *application/x-www-urlencoded*. Il formato *application/x-www-form-urlencoded* permette di trasmettere i parametri all'interno del contenuto di una request, usando la stessa sintassi che serve a comunicare i parametri nella URL.

```
var bodyParser = require('body-parser');
app_backend.use(bodyParser.urlencoded({ extended: false }));
```

Essendo questo uno dei due file che verranno effettivamente eseguiti si stampano le porte assegnate al backend e al frontend:

```
brainjobs-backend : node index.js

Web Server started.
API started on: 8082
Static index on: 8081
```

Passiamo ora al file con le api vere e proprie. Come abbiamo visto viene caricato passandogli come parametri l'app istanziata da *express* (che gode dei metodi *.get()* e *.post()*) e il database generato mediante *lowdb*. All'inizio avremo quindi:

```
module.exports = function (app, db) {
  ...
};
```

Sul primo endpoint, */brainjobs-backend/api/v1/jobs*, abbiamo innanzitutto il salvataggio di una richiesta, mediante il metodo POST. Si ha accesso all'oggetto richiesta con *req* e risposta con *res*

```
app.post("/brainjobs-backend/api/v1/jobs", (req, res) => {
  ...
});
```

Viene richiesto di aggiungere ai dati inseriti dall'utente un campo *job_id* univoco, la data e lo stato della richiesta. Viene quindi usata una libreria aggiuntiva, *uuid*, nella versione 4, <https://github.com/kelektiv/node-uuid>, per generare degli id univoci, infatti *uuid* sta per *Universally Unique Identifier*.

```
var uuidv4 = require('uuid/v4');
var job_id = uuidv4();
```

Per la data di creazione (campo *created_at*) viene usata la classe *Date* il cui risultato viene parsato in un formato “human”, composto di giorno, mese e anno. Il metodo *padstart(n, c)* trasla la stringa fino al raggiungimento di *n* caratteri aggiungendo in testa la giusta quiantità di caratteri *c*. Quindi prende mediante il metodo *getDate()* il giorno del mese e, se di una sola cifra, ci emtte davanti uno 0. Poi con il meotodo *.getMonth()* si ottiene il mese (ma partendo da 0, che rappresenta Gennaio) al quale viene aggiunto 1, procedendo poi come per il giorno. Per l’anno si usa banalmente il metodo *getFullYear()*:

```
var today = new Date();
var dd = String(today.getDate()).padStart(2, '0');
var mm = String(today.getMonth() + 1).padStart(2, '0');
var yyyy = today.getFullYear();
var created_at = dd + '/' + mm + '/' + yyyy;
```

per quanto riguarda il campo *status* viene aggiunto il valore *created* in quanto è l’unico scenario di successo di questo progetto:

```
var status = 'created';
```

Come vedremo dopo il backend non è in diretto contatto con il frontend ma si ha di mezzo un gateway che nel nostro casso passa una stringa contenente un json al backend. Si ha quindi il metodo *JSON.parse(string)* per ottenere un oggetto json con i dati della richiesta (quindi solo quelli del campo *req.body* che per comodità vengono assegnati ad una variabile con lo stesso nome):

```
req.body = JSON.parse(Object.keys(req.body)[0]);
```

Viene quindi definito l'oggetto con la richiesta sfruttando appunto *req.body* e i tre campi sopra definiti:

```
var request = {
  user_id: req.body.user_id,
  title: req.body.title,
  language: req.body.language,
  framework: req.body.framework,
  dataset: req.body.dataset,
  dataset_datatype: req.body.dataset_datatype,
  model: req.body.model,
  status: status,
  created_at: created_at,
  job_id: job_id
};
```

SI provvede poi al salvataggio della richiesta nel database sfruttando il metodo *.get(string)* di lowdb che seleziona il campo *request* impostato di default, con il metodo *.push(object)* inserisce l'oggetto json e con il metodo *.write()* scrive nel file:

```
db.get('requests').push(request).write();
```

Viene settato quindi l'header della risposta aggiungendo i tre parametri creati, e si aggiunge l'intestazione *'Access-Control-Allow-Origin': '*'*, usata per determinare se è possibile accedere alla risorsa partendo dal contenuto che opera nell'origine corrente:

```
res.set({
  'Access-Control-Allow-Origin': '*',
  'Status': status,
  'Created_at': created_at,
  'Location': job_id
});
```

Infine viene inviata la risposta con status code pari a 201, indicando specificatamente il “path” *job_id* contenuto nell’header (???) inviando un json contenente i vari campi (??).

```
res.status(201).location(job_id).json(request);
```

Sempre sullo stesso endpoint si ha anche la query mediante lo *user_id*, usando il metodo GET:

```
app.get("/brainjobs-backend/api/v1/jobs", (req, res) => {
  ...
});
```

Viene settato quindi l’header della risposta con l’intestazione *'Access-Control-Allow-Origin': '*'*,

```
res.set({
  'Access-Control-Allow-Origin': '*'
});
```

E si passa quindi alla query al database, mediante prima il *.get(string)* per selezionare il giusto campo, poi il metodo *.filter(string)* per effettuare la

query selezionando solo quelli che hanno una certa condizione, e la *.value()* che restituisce i valori, e all'invio, in formato json della ripsota, con il metodo *.json(string)* La condizione viene rappresentata mediante la richiesta che il campo *user_id* corrisponda allo *user_id* presente in *req.query*, che sfrutta l'url (da notare il "?") *jobs?user_id=user_id*:

```
res.json(db.get('requests')
  .filter({ user_id: req.query.user_id }).value()));
```

Abbiamo infine l'altra GET, quella per ottenere la richiesta corrispondente ad un *job_id* sull'endpoint dedicato: */brainjobs-backend/api/v1/jobs/:job_id*. Il principio è simile a quello dell'altra query solo che useremo il metodo *.find(string)* per trovare l'unica richiesta co quel dato *job_id*. Inoltre la query verrà effettuata mediante il *job_id* passato nell'endpoint(con il ":") al quale quindi non si accede più con il metodo *query* bensì con quello *params*, che ritorna il nome del parametro quando presente:

```
app.get("/brainjobs-backend/api/v1/jobs/:job_id", (req, res) =>{
  res.set({
    'Access-Control-Allow-Origin': '*'
  });
  res.json(db.get('requests')
    .find({ job_id: req.params.job_id }).value());
});
```

Capitolo 4

Gateway

Nell'ultimo punto del progetto si introduce il concetto di API Gateway. L'API Gateway è un componente che, fra le sue funzionalità, maschera la presenza di uno o più backend alle applicazioni di frontend, occupandosi di esporre una API univoca e di aggregare, filtrare e comporre richieste/risposte per/da applicazioni di backend. Nel nostro caso, semplificato, l'API Gateway può replicare per semplicità la stessa API esposta dal backend ma il suo compito sarà quello di utilizzare internamente un REST Client per inoltrare le richieste del frontend al backend e restituirne le risposte.

Nel complesso si ottiene quindi la seguente architettura generale:

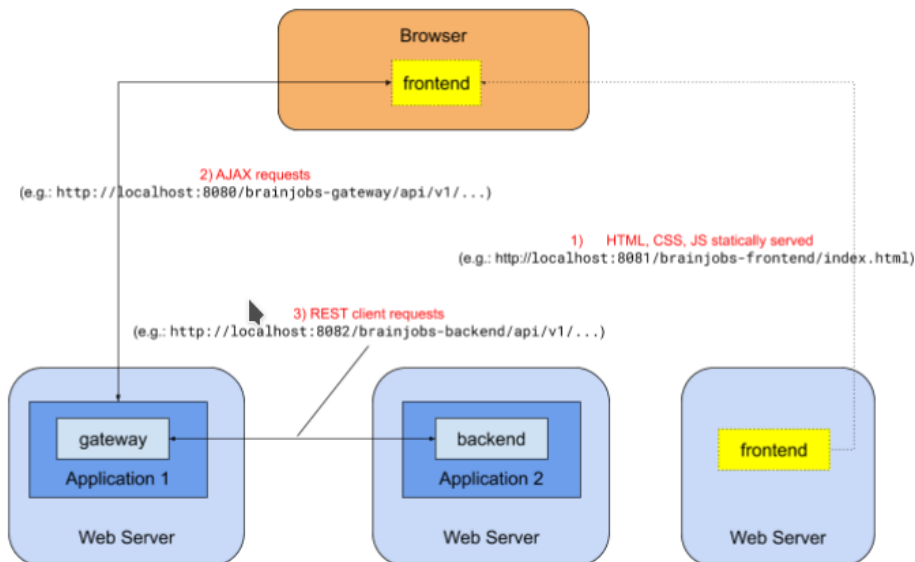


Figura 4.1: architettura finale del progetto

Abbiamo la seguente struttura:

```
brainjobs-gateway
  api
    index.js
  index.js
  package.json
  package-lock.json
```

Partiamo, come nel caso del backend, dall'*index.js* presente subito nella cartella dedicata al gateway. Creiamo anche qui una istanza con *express*. Viene anche usato *body-parse*, <https://github.com/expressjs/body-parser>, che parse il body della richiesta in un middleware, raggiungibile mediante *req.body*, usando il metodo *.urlencoded(options)* per il parsing di dati nel formato *application/x-www-urlencoded* e si mette l'istanza in ascolto sulla porta 8080.

SI ha una differenza rispetto al backend, infatti qui si inizializza una costante che istanzia un client HTTP basato su *promise*, i quali oggetti rappresentano l'eventuale completamento (o fallimento) di un'operazione asincrona: *axios*, <https://github.com/axios/axios>. L'istanza di *express* e quella di *axios* vengono passati come parametro all'*index.js* contenuto nella cartella */api*. Dobbiamo implementare un gateway che effettua un semplice reindirizzamento.

Cominciamo con l'inoltro di tutte le richieste GET. Si usa il metodo *.get()* dell'istanza di *express*. Subito viene definita una variabile contenente l'url a cui deve essere reindirizzata la GET, ovvero il path attuale della richiesta (recuperabile con *req.originalUrl*) e usando il metodo *replace(a, b)* delle stringhe con *brainjobs-backend* al posto di *brainjobs-gateway*:

```
module.exports = function (app, axios) {

  app.get("/brainjobs-gateway/api/v1/*", (req, res) => {

    var redirect = 'http://localhost:8082' +
      req.originalUrl.replace('gateway', 'backend');
    ...
  })
}
```



```
});  
}
```

Entra quindi in gioco *axios*, a cui vengono passate le opzioni riguardanti il metodo della richiesta, in questo caso *get*, l'url a cui fare la richiesta, ovvero l'url del backend che abbiamo appena creato, e il formato della risposta del server, in questo caso *json*. Con il metodo *.then(function(response))* si esegue una funzione specifica, nel nostro caso si setta l'header della risposta con *Access-Control-Allow-Origin **, permettendo a tutti di accedere alla risorsa partendo dal contenuto che opera nell'origine corrente. Infine con *res.json(object)* viene inviata la risposta in formato json. *axios* richiede obbligatoriamente anche di implementare una funzione in caso di errore, e nel nostro caso consiste nell'invio di un json contenente la specifica dell'errore:

```
axios({  
  method: 'get',  
  url: redirect,  
  responseType: 'json'  
})  
  .then(function (response) {  
  
    res.set({  
      'Access-Control-Allow-Origin': '*'  
    });  
  
    res.json(response.data);  
  
  })  
  .catch(function (error) {  
    console.log(error);  
    res.json({'error': ''+error});  
  });  
});
```

Nell medesimo modo reindirizziamo anche le richieste post, impostando il metodo come *post*, impostando i dati da inviare come quelli contenuti nel body della richiesta, con *req.body*, e settando nell'header il parametro *application/x-www-form-urlencoded* sopra spiegato:

```
app.post("/brainjobs-gateway/api/v1/*", (req, res) => {

  var redirect = 'http://localhost:8082' +
    req.originalUrl.replace('gateway', 'backend');
  var forward_data = req.body;

  axios({
    method: 'post',
    url: redirect,
    responseType: 'json',
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded'
    },
    data: forward_data
  })
  .then(function (response) {

    res.set({
      'Access-Control-Allow-Origin': '*'
    });

    res.json(response.data);

  })
  .catch(function (error) {
    console.log(error);
    res.json({'error': ''+error});
  });

});
```

Notiamo come sia nella GET che nella POST viene usata la wildcard “*” nell'url per permettere un reindirizzamento completo.

In fase di esecuzione si avrà:

```
brainjobs-backend: node index.js  
Gateway started on: 8080
```

Capitolo 5

Esecuzione del Progetto

Per eseguire il progetto bisogna innanzitutto verificare che le 3 porte 8080, 8081 e 8082 siano libere. Bisogna poi spostarsi nelle cartelle `brainjobs-backend` e `brainjobs-gateway` e in entrambe eseguire il comando `npm install`:

```
brainjobs-backend: npm install
  audited 167 packages in 1.767s
  found 0 vulnerabilities

brainjobs-backend: cd ../brainjobs-gateway
brainjobs-gateway: npm install
  audited 163 packages in 1.611s
  found 0 vulnerabilities
```

per installare le eventuali dipendenze elencate nel file `package.json` nella cartella `node_modules`. A questo punto bisogna contemporaneamente eseguire i due comandi `node index.js` all'interno delle cartelle `brainjobs-backend` e `brainjobs-gateway`. Un comando comodo può essere, per non dover aprire più terminali:

```
cd brainjobs-backend ; \
node index.js & ; \
cd .. ; \
cd brainjobs-gateway ; \
node index.js
```

Una seconda alternativa può essere quella di usare un *process manager* per nodejs come **pm2**, <https://github.com/Unitech/PM2>. Dopo averlo installato con:

```
npm install -g pm2
```

l'uso è simile alla sequenza di comandi data sopra, con la differenza che sarà pm2 ad occuparsi di eseguire il tutto, permettendo eventualmente anche il controllo dei vari server (nominati con l'opzione *-name*) in esecuzione mediante logs, interfaccia web o monitor tty. Si ha quindi, per esempio:

```
cd brainjobs-backend && \  
  pm2 start index.js --name backend && \  
cd .. && \  
cd brainjobs-gateway && \  
  pm2 start index.js --name gateway && \  
pm2 logs
```

Capitolo 6

Docker

Docker è un progetto open-source che automatizza il deployment (consegna o rilascio al cliente, con relativa installazione e messa in funzione o esercizio, di una applicazione o di un sistema software tipicamente all'interno di un sistema informatico aziendale) di applicazioni all'interno di contenitori software, fornendo un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo di Linux. Docker utilizza le funzionalità di isolamento delle risorse del kernel Linux come ad esempio cgroups e namespaces per consentire a "container" indipendenti di coesistere sulla stessa istanza di Linux, evitando l'installazione e la manutenzione di una macchina virtuale. Docker permette di esporre un set di porte desiderato.

Realizzare un container per questo progetto è abbastanza veloce, si crea un *dockerfile* contenente innanzitutto l'immagine da cui si vuole partire. Per nodejs, su dockerhub, è disponibile l'immagine ufficiale (https://hub.docker.com/_/node). Si crea poi una directory su cui lavorare nel container e si copiano le cartelle del progetto, si eseguono i comandi per installare le dipendenze, si espongono le tre porte necessarie e infine si definisce il comando da eseguire alla fine.

Un esempio di dockerfile, usando il metodo di esecuzione con *pm2* è:

```
FROM node:10

# Create app directory
WORKDIR /usr/src/app

# Copy folders
COPY brainjobs-backend ./brainjobs-backend
COPY brainjobs-frontend ./brainjobs-frontend
COPY brainjobs-gateway ./brainjobs-gateway

# Install dependences
RUN cd brainjobs-backend && npm install
RUN cd brainjobs-gateway && npm install

# Install Process Manager
RUN npm install -g --silent pm2

# Open ports
EXPOSE 8080 8081 8082

# Start
CMD cd brainjobs-backend && \
    pm2 start index.js --name backend && \
    cd .. && \
    cd brainjobs-gateway && \
    pm2 start index.js --name gateway && \
    pm2 logs
```

Si ha inoltre un file chiamato *.dockerignore* dove si elenano i nomi dei file o delle directory che devono essere ignorati nella fase di COPY. Nel nostro caso le cartelle *node_modules* in quanto verranno generate da *npm install*, i log e il database in json di *lowdb*, in quanto viene creato automaticamente se assente e settato con dei parametri di default specifici.

Si ha quindi un semplice file *.dockerignore*:

```
node_modules
npm-debug.log
db.json
```

Controllando di essere nella directory del dockerfile e del *.dockerignore*, buildiamo la nostra immagine con il comando:

```
docker build -t brainjobs .
```

Questa fase impiegherà diversi minuti, soprattutto a causa dell'installazione di *pm2*. Infine eseguiamo l'immagine, specificando che vogliamo lo stesso numero di porta tra quello della macchina fisica e quello esposto dal container:

```
docker run -p 8080:8080 -p 8081:8081 -p 8082:8082 brainjobs
```