UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI

INGEGNERIA ELETTRICA E MATEMATICA APPLICATA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Web Scraping per l'estrazione automatizzata di informazioni da sorgenti Web

Relatore:
Ch.mo Prof.
Pierluigi RITROVATO

Candidato:
Claudio Salvatore DI MAURO
Matr.: 0612704811

ANNO ACCADEMICO 2019-2020

Ai sognatori, agli speranzosi,
ai fuoricorso.

ABSTRACT

Il presente elaborato ha l'obiettivo di sintetizzare quello che è stato l'operato di tirocinio effettuato presso l'Azienda **Engineering Ingegneria Informatica S.p.A.** e che ha visto come oggetto il **Web Scraping**, ossia l'estrazione automatizzata di dati da sorgenti web.

Le aspettative accademiche e aziendali per questo tirocinio prevedevano che si realizzasse uno scraper che potesse interfacciarsi in maniera adatta alla piattaforma **MedITech** e che ne alimentasse i cataloghi in maniera automatica.

Il primo passo fatto per effettuare l'estrazione dei dati, è stato quello di individuare gli elementi rilevanti per la ricerca dalle sorgenti web monitorate, per poi formalizzare il tutto secondo il modello noto, definito dai cataloghi che si è andati a popolare. Mediante l'utilizzo di API e di piattaforme specifiche si è provveduto alla configurazione dei processi e allo sviluppo della logica di estrazione.

Il lavoro è stato sviluppato su vari step, ognuno dei quali ha previsto di aggiungere al progetto nuove funzionalità che andassero ad incrementare quelle di base (da garantire per la corretta conclusione dello stesso).

Si è deciso di impostare il tirocinio come lavoro di gruppo sviluppato da due componenti.

Il progetto è partito andando a sfruttare Java come linguaggio di programmazione, con particolare attenzione alla libreria JSoup, la quale presenta i metodi e le funzionalità necessarie ad effettuare lo scraping.

È stato richiesto di utilizzare tecnologie come **Docker** per poter lavorare: si è utilizzata questa piattaforma per mantenere in esecuzione il database su cui sono stati salvati i dati di interesse pervenuti dallo scraping e per formalizzare tutto il progetto secondo uno standard – che va sotto il nome di "immagine" – per la piattaforma stessa.

Si è scelto di adottare una architettura a microservizi, in modo da poter lavorare esponendo delle **REST APIs** che l'utente finale può invocare per operare lo scraping.

Usando **Java**, il modo migliore per implementare tale soluzione è stato quello di utilizzare il framework **Spring**, con particolare utilizzo della sua estensione **Spring Boot**. Tale estensione lavora in modo tale da facilitare al programmatore che la utilizza tutta una serie di configurazioni che altrimenti andrebbero effettuate manualmente.

Si è deciso di utilizzare un database di appoggio e nello specifico la scelta è ricaduta su **MongoDB**, un database non relazionale (quindi **NoSQL**) che per la sua versatilità è risultato perfettamente adatto agli scopi del progetto. MongoDB è infatti schemeless, cioè per i documenti contenuti al suo interno non c'è bisogno di adottare uno schema di lavoro ben preciso ma ogni documento può avere una struttura a sé stante. Data l'eterogeneità delle sorgenti su cui lo scraping può essere effettuato, Mongo è risultato essere perfettamente compatibile con le necessità di lavoro.

Il software è partito con uno sviluppo basato sullo scraping "statico", ossia con del codice pensato apposta per una e una sola determinata sorgente Web. Con il tempo si è evoluto fino a diventare uno scraping dinamico o, per lo meno, semi-dinamico, dato che oltre alle entità da "raschiare" sono stati definiti anche dei patterns che permettono di selezionare il codice dal **DOM** della pagina di cui si vuole fare lo scraping e operare sulla base di questo.

Secondo questo modello dinamico, l'utente può scegliere di analizzare una qualsiasi sorgente di sua scelta, a patto che riesca ad avere una conoscenza profonda e consapevole della struttura HTML (quindi del DOM, appunto), della pagina su cui sta andando a lavorare.

La soluzione dei pattern è sembrata quella che meglio si sposasse con ciò che si necessitava; di fatti, ogni pattern ha una struttura di base presentata all'utente sotto forma di un documento **JSON** con il classico schema chiave-valore, dove il proprio il valore degli argomenti del JSON deve essere opportunamente modificato per ottenere il comportamento desiderato. Le modifiche consistono nella trascrizione dei tag e degli attributi HTML idonei alla realizzazione dello scraping di ciò che si intende analizzare e ottenere.

GitHub si è dimostrato essere uno strumento fondamentale per il lavoro di gruppo. Ha permesso lo sviluppo delle varie parti del codice in maniera separata e grazie al sistema di versioning su cui è basato, cioè **Git**, ha permesso di scrivere codice senza imporre un confronto continuo tra gli sviluppatori.

Per questo progetto, ogni scelta è stata presa a valle di diverse considerazioni e di un confronto costante tra le parti, al fine di poter ottenere un risultato ottimale secondo quanto richiesto dal piano formativo.

Indice dei contenuti

Indice delle figure

Indice degli snippet

CAPITOLO 1: INTRODUZIONE

Il capitolo introduce al lavoro sviluppato, evidenziando la logica di progetto e al contesto in cui questo è stato realizzato. Si accenna alla soluzione proposta e a tutte le tecnologie e le strutture che ne hanno permesso la realizzazione.

1.1 Il progetto

1.1.1 Il tirocinio in azienda e la piattaforma MedITech

Il presente elaborato di tesi di laurea triennale nasce dalla volontà di approfondire e analizzare il complesso e sempre più eterogeneo mondo del **Web Scraping**.

È utile delineare un quadro ben definito dell'iter che ha portato alla realizzazione del progetto finale. Grazie a un'esperienza svolta sul campo, offerta dall'avvio di un tirocinio curricolare avviato dall'**Università di Salerno** presso la **Engineering Ingegneria Informatica S.p.a.**, è emersa la possibilità di aprire un focus scelto come caso di studio per il presente lavoro. Il fine ultimo di questo progetto è quello di realizzare un software che in maniera automatizzata sia capace di estrarre informazioni di taluna rilevanza da specifiche sorgenti Web.

L'opportunità è nata grazie ad una proposta di tirocinio presentata dalla suddetta Azienda, che è una dei principali attori della trasformazione digitale di aziende e organizzazioni pubbliche e private, con un'offerta innovativa per i principali segmenti di mercato. Il Gruppo in questione è attivo su diversi fronti nel campo delle tecnologie informatiche ed è parte integrante del consorzio **MedITech**, che si pone come "trasportatore" di nuove tecnologie verso le industrie 4.0.

Nello specifico, la piattaforma MedITech si presenta come tramite tra l'Azienda e il progetto da sviluppare. Infatti, quello che si richiede nelle specifiche previste dall'offerta formativa del tirocinio è di andare a sfruttare la tecnologia del Web Scraping per popolare dei cataloghi ospitati sulla piattaforma di cui sopra, i quali sono dei contenitori di informazioni rilevanti di cui in una fase iniziale non è dato conoscere la natura e che dovrebbero essere riempiti per mezzo di servizi in grado di "osservare" una lista di sorgenti pubbliche da cui estrarre le informazioni necessarie.

1.1.2 La logica di progetto

Al fine di operare in maniera idonea, è stato richiesto di strutturare e definire una buona logica progettuale, che permetta un corretto interfacciamento con i cataloghi menzionati al paragrafo precedente.

Nello specifico, è stato richiesto dall'Azienda che i dati estratti vengano immagazzinati secondo il modello noto dei database MedITech. È stato altresì necessario definire un formato strutturato per la rappresentazione dei dati.

Nei prossimi capitoli sarà spiegata la struttura rappresentativa dei dati, la tipologia di database utilizzati e tutte le tecnologie ad essi associati.

1.2 Approccio alla soluzione

1.2.1 Architettura, linguaggio e persistenza dei dati

Il progetto è stato strutturato su varie fasi, ognuna delle quali, dopo un confronto con i tutor aziendali, ha visto nascere nuove features che man mano lo hanno arricchito.

Si è partiti dall'idea iniziale di cui si è già parlato nei paragrafi precedenti e si è iniziato il lavoro andando ad analizzare quelle che sarebbero potute essere le soluzioni migliori per portarlo avanti.

Il Web Scraping è una tecnologia fruibile in tanti linguaggi di programmazione, dato che molti di essi (quali Java, Java Script, React, NodeJS, Python, Angular, ecc.) offrono delle buone librerie che ne permettono la messa in pratica. Tra i vari linguaggi la scelta è ricaduta su **Java**, questo perché oltre ad avere una libreria molto ricca e un supporto molto ampio da parte delle community di sviluppatori, gode anche di un *framework*[1] (vedremo nel seguito quale) che lo rende estremamente idoneo a lavorare con le API REST, altro elemento chiave delle scelte progettuali vagliate per il presente lavoro: si è infatti scelto, come si leggerà in maniera più approfondita nel paragrafo 2.2.1, di adottare una **architettura a microservizi**, precisamente un *pattern architetturale* basato sulla divisione dell'applicazione in tanti piccoli "frammenti", ognuno dei quali indipendente dagli altri, che sviluppa precise funzionalità.

[1] **Framework**, in informatica, e più nello specifico nello sviluppo software, rappresenta una architettura di supporto (spesso implementazione di un particolare *design pattern*) sulla quale un *software* può essere progettato e realizzato, facilitandone lo sviluppo da parte del programmatore.

Un’altra delle scelte effettuate è stata quella di utilizzare una tipologia di database che sempre più si sta affermando nell’ambito dello sviluppo software moderno, vale a dire i database **NoSQL**. Si tratta di database non relazionali, cioè non basati – come si è sempre stati abituati a vedere – su tabelle, ma su **documento**, quale elemento centrale di queste basi dati. Ogni documento, tipicamente un file JSON-like, contiene le informazioni relative ad una delle parti da salvare: un insieme di documenti compone il database vero e proprio. Questo argomento e lo specifico DB utilizzato sono ampiamente approfonditi nel paragrafo 2.2.4.

CAPITOLO 2: ASPETTI SALIENTI E TECNOLOGIE ABILITANTI

Nel capitolo si introduce il Web Scraping, al modo in cui esso è stato utilizzato nel progetto e alla sua rilevanza tecnologica. Degli esempi ne chiariscono il funzionamento e se ne evidenziano le possibili problematiche.

In questo capitolo saranno anche analizzate una ad una le tecnologie utilizzate per realizzare il progetto, facendo un focus particolare sulle caratteristiche di ognuna e sul perché sono state scelte.

2.1 Il Web Scraping

2.1.1 Definizione, utilizzo e rilevanza tecnologica

Il **Web Scraping** [1] (dall'inglese *to scrape*, cioè "raschiare"), è una tecnica di estrazione automatizzata di dati da siti Web.

Tale tecnica è operata per mezzo di strumenti software ed è ampiamente utilizzata dai motori di ricerca nell'indicizzazione delle pagine Web, effettuata nella maggioranza dei casi per mezzo di **bot**[2].

Con il Web Scraping è possibile raccogliere tutta una serie di informazioni di varia natura. In particolare, si distinguono due macro tipologie di scraping [2]: lo **scraping manuale**, che consiste nel fare il classico "copia e incolla" dai siti di interesse per estrapolare le informazioni necessarie, ovviamente con un dispendio temporale altissimo tanto da non essere mai utilizzato per l'estrazione di grandi quantità di dati; poi c'è lo **scraping automatico**, che invece utilizza software o algoritmi e che, ovviamente, consente una raccolta dati molto più veloce e flessibile, offrendo anche la possibilità di una ricerca contemporanea su più pagine.

Per lo scraping automatico, si distinguono fondamentalmente tre **tecniche**:

- **Parser HTML**: un parser è, grossolanamente, un "traduttore" che permette di convertire del testo in una nuova struttura. Nel caso specifico dell'analisi HTML

[2] **Bot** è l'abbreviazione inglese di *robot*. In informatica con il termine *bot* si intende un software che esegue azioni automatiche, talvolta simulando il comportamento umano.

per lo scraping, il software legge le informazioni contenute nel **DOM**[3] utilizzando la visualizzazione lato client (si simulano le interazioni e le richieste che effettua un browser Web);

- **Bot:** la navigazione Web e la raccolta dati avvengono in maniera automatica, con processi che simulano le interazioni umane con i siti Web;
- **Text:** la riga di comando, tramite i comandi **Unix Grep** o tramite **cURL**, rappresenta un ulteriore metodo di scraping che però richiede un lavoro maggiore (bisogna scrivere ogni volta a mano i codici da riga di comando), rispetto all'utilizzo di un software.

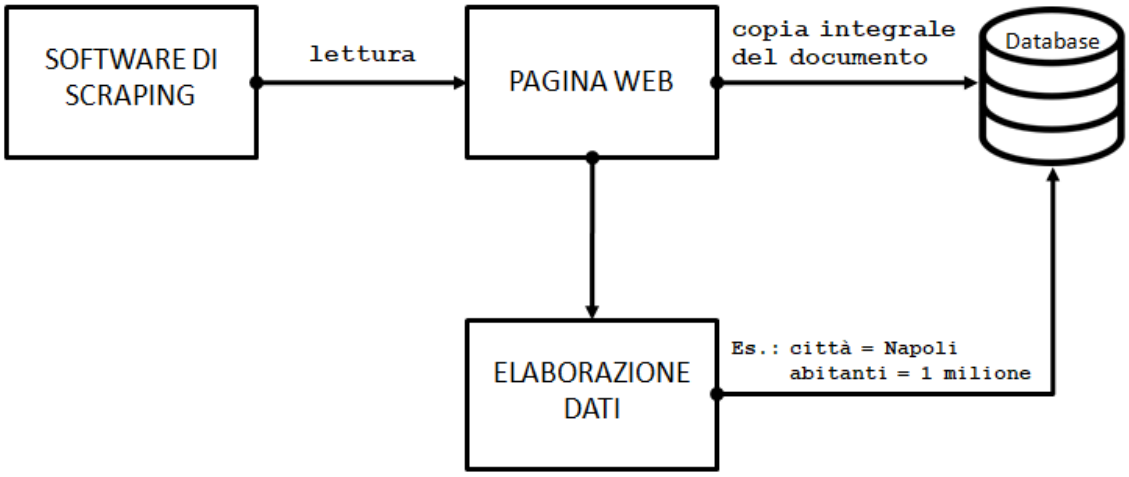

Fig. 1 - Funzionamento del Web Scraping

Dovrebbe risultare chiaro, a questo punto, quanto il Web Scraping abbia una rilevanza tecnologica non indifferente, dal momento in cui esso consente di ottenere in maniera quasi istantanea centinaia o addirittura migliaia di informazioni, che vanno da dati di contatto a informazioni relative a potenziali competitors e così via. Per le aziende, fare scraping è un'attività fondamentale in quanto una buona analisi e quindi un buon *harvesting dei dati* permette loro di effettuare confronti diretti tra i propri prodotti e quelli delle aziende avversarie, potendo poi applicare delle procedure di "difesa" per trovarsi in vantaggio sulla concorrenza.

[3] **DOM** [3], *Document Object Model*, è un'interfaccia di programmazione standardizzata per la strutturazione di documenti HTML e XML; il suo scopo è quello di facilitare l'accesso ai componenti di un progetto web.

2.1.2 Aspetti legali, problematiche e prevenzione

È fondamentale sapere che lo scraping **non sempre è legale**. Tale pratica deve infatti tenere conto dei diritti d'autore a cui un sito Web sottostà, pertanto eventuali dati protetti da copyright non possono essere riprodotti altrove o commercializzati. Lo scraping è quindi legale in tutti i casi in cui i dati estratti sono liberamente accessibili a terzi sul Web.

Altro aspetto di cui tener conto, e questo è prettamente etico, riguarda la quantità di dati che si va a leggere tramite lo scraping nel breve termine. Uno scraping intenso, infatti, può portare danni al server del sito scansionato a causa dell'intenso traffico che va a generare.

Uno dei grossi problemi legati allo scraping riguarda lo **spam**: viene spesso utilizzato dagli *spammers* per raccogliere indirizzi e-mail e inviare a questi destinatari messaggi indesiderati.

Proteggere il proprio sito dallo scraping si può ed esistono varie tecniche per farlo: è possibile configurare un firewall, bloccare gli indirizzi IP dei bot, salvare dati sensibili sotto forma di CSS o come immagine.

2.1.3 Esempi comuni di Web Scraping

È già stato analizzato nei paragrafi precedenti di come i **crawler**[4] dei motori di ricerca facciano largo uso del Web Scraping per ottenere informazioni dai siti ed indicizzarli in maniera adeguata.

Tutti i giorni capita di fare una ricerca su Google e di ottenere particolari informazioni. È bene sapere che tali informazioni spesso sono frutto diretto del "raschiamento" di un sito.

[4] Un **crawler** [4] o *spider* è un software che analizza i contenuti di una rete o di una base dati in modo metodico e automatizzato, tipicamente per conto di un motore di ricerca. Un crawler è un esempio di bot.

Cercando su Google “Fatturato Fiat”, quello che verrà mostrato come primissima informazione è il seguente box:

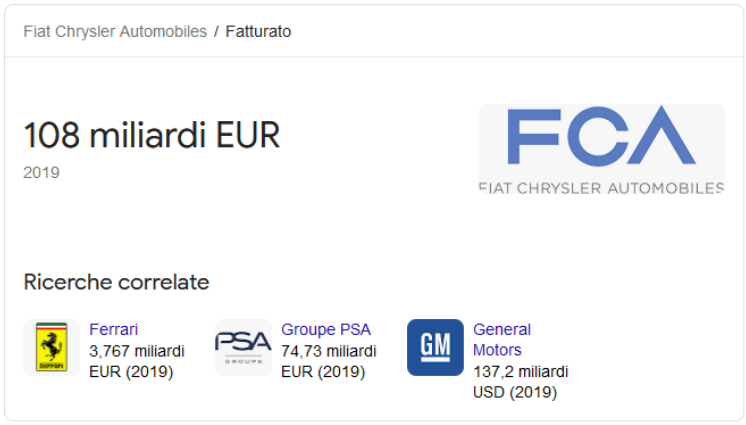

Fig. 2 - Scraping fatturato FCA

O, ancora, cercando “Abitanti Napoli”, ci verrà fornito questo:

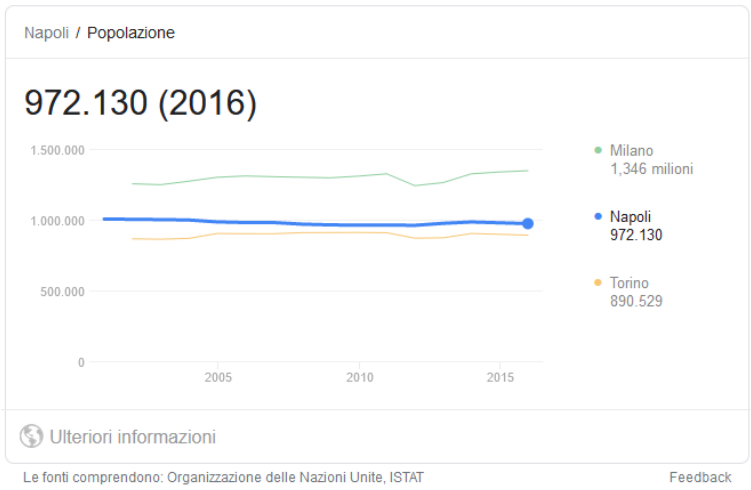

Fig. 3 - Scraping abitanti di Napoli

In entrambi i casi mostrati, si ha una serie di informazioni riassuntive che, a partire dalla ricerca effettuata, vengono mostrate in maniera rapida. Nel secondo caso, come si vede, sono anche identificate le fonti da cui i dati sono stati ricavati (banche dati di ONU e ISTAT).

L’utilizzo “intelligente” che Google fa dello scraping, permette di avere rapidamente le informazioni di cui si ha bisogno, fornendo le più aggiornate possibili: se si

vuole conoscere il fatturato di FCA, non interessa navigare su un sito e scorrerlo fino ad arrivare all'informazione desiderata, ma è molto più comodo averla direttamente.

Si noti che questa funzionalità è anche implementata in Google Assistant (quindi sia nell'app mobile che sul dispositivo Google Home); si provi allora a dire "*hey Google, qual è il fatturato di Fiat?"*: la risposta sarà la stessa mostrata in *Fig.2*.

2.2 Scelte progettuali e tecnologie utilizzate

2.2.1 Architettura a microservizi

Per il progetto di tirocinio si è scelto di adottare una **architettura a microservizi** [5], ossia quell'approccio architetturale che, a differenza dell'architettura monolitica, prevede di modularizzare una applicazione nelle sue funzioni base. [6] Ciascuna funzione è detta *servizio* e viene implementata e compilata in modo totalmente indipendente dalle altre. I singoli servizi devono poter funzionare o guastarsi senza compromettere gli altri.

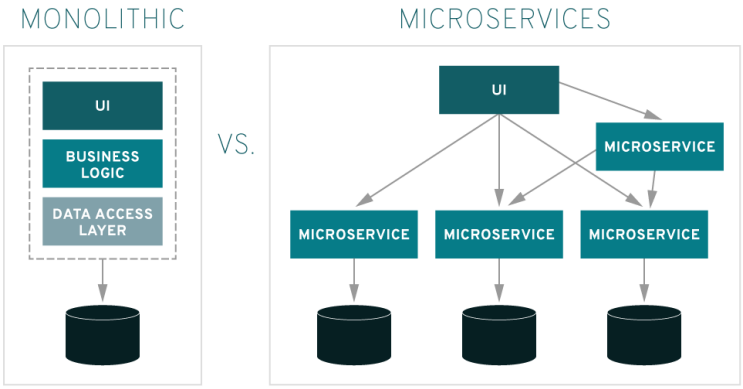

Fig. 4 - Architettura monolitica vs architettura a microservizi

Il concetto di *pattern a microservizi* prevede che i singoli servizi siano incentrati su uno specifico processo aziendale e seguano un determinato protocollo di comunicazione: è fondamentale che ogni microservizio possa comunicare con gli altri per avere una maggiore tolleranza ai guasti.

Basandosi su architetture distribuite, i microservizi consentono di ottenere uno sviluppo software più efficiente, con la possibilità che più microservizi vengano svilup-

pati in contemporanea da più sviluppatori che lavorano simultaneamente, ognuno dei quali si occupa solo di uno o, al più, pochi servizi.

2.2.2 Java

Il linguaggio di programmazione scelto per implementare il progetto è stato Java. Insieme a linguaggi come Python, JavaScript e NodeJS, si presta molto bene a questo tipo di operazioni, in quanto dotato di librerie e strutture che ne facilitano la scrittura del codice.

Fig. 5 - Logo Java

Java è un linguaggio di programmazione di alto livello fortemente orientato agli oggetti: in Java tutto è una classe. Sviluppato nel 1995 dalla **Sun Microsystems**, è ad oggi uno dei linguaggi più usati e supportati al mondo, che conta milioni di programmi, applicazioni mobile e web app sviluppate.

La cosa fondamentale da conoscere su Java è la filosofia che c'è dietro di esso: *write once, run anywhere*, ossia "scrivi una volta, esegui ovunque". Java è infatti un linguaggio che basa il suo funzionamento sulla **JVM** (*Java Virtual Machine*), ossia una macchina virtuale che, quando installata, permette di far girare i software su qualsiasi piattaforma: una volta compilato il programma, il bytecode può essere eseguito su qualsiasi sistema operativo (Linux, MacOS, Windows,…).

Essendo sempre rimasto al passo coi tempi, Java non si è privato di una libreria per il Web Scraping. Quella utilizzata in questo progetto è **JSoup** [7]**,** un parser HTML fortemente incentrato sulla flessibilità e la facilità d'uso. Consente un'estrazione di dati

dall'HTML sfruttando i metodi del DOM e i selettori **CSS**[5]. È ovviamente un progetto open-source, il cui codice è disponibile su **GitHub**[6].

2.2.3 Spring Boot e i servizi REST

Fig. 6 - Logo Spring e logo Spring Boot

Spring, e in particolare il suo derivato **Spring Boot**, è un framework Java sviluppato per permettere la programmazione mediante microservizi. In particolare tale framework consente di sfruttare questo pattern architetturale facendo uso di **Rest API** [8] che, una volta "esposte" e invocate, permettono l'interazione tra sistemi indipendenti, sfruttando tutte le potenzialità del protocollo **HTTP**[7].

Le richieste REST vengono effettuate per tramite di quelli che si chiamano *verbi HTTP* (o *metodi HTTP*) e ogni richiesta specifica nella sua intestazione un determinato verbo (e.g. GET, PUT, POST, DELETE, …).

Il client effettua la richiesta al server HTTP specificando il servizio che gli serve (tramite la API) e come glielo deve fornire (tramite i metodi). Il server può rispondere in determinati modi, a seconda che il servizio o la risorsa richiesti siano disponibili, non disponibili, richiesta non valida, ecc. La più comune (e nota a tutti) è la risposta con codice **404**, con la quale il server specifica che la risorsa richiesta non è stata trovata (capita spesso che navigando su un sito ci si può trovare di fronte al messaggio "404 Page Not Found": questo messaggio è la risposta del server).

---

[5] **CSS**, *Cascading Style Sheets*, è un linguaggio utilizzato per la formattazioni dei documenti ipertestuali, quali HTML, XHTML e XML. Il CSS permette di separare i contenuti di un documento ipertestuale dalla sua formattazione, al fine di permettere una programmazione più chiara e facilmente riutilizzabile.

[6] **JSoup**, codice sorgente disponibile su: https://github.com/jhy/jsoup/

[7] **HTTP**, *HyperText Transfer Protocol*, protocollo a livello applicativo usato come principale sistema per la trasmissione d'informazioni sul Web ovvero in un'architettura tipica client-server.

2.2.4 MongoDB

Fig. 7 - Logo MongoDB

[9] **MongoDB** è un database nato da un progetto open-source, sviluppato a partire dal 2007[8] e che sempre più si sta facendo strada come tecnologia di alto livello a supporto degli sviluppatori.

In particolare, Mongo è un database *orientato ai documenti* e classificato come *NoSQL*, in cui la persistenza dei dati non è basata su tabelle, campi e record (come nel caso dei database relazionali – e.g. *MySQL*), ma su file di tipo **JSON**[9] con uno schema dinamico che in ambiente Mongo è detto **BSON** (acronimo di *Bynary JSON*). All'interno di questi JSON Binari i dati vengono salvati secondo uno schema chiave-valore, in cui la "chiave primaria" è definita come **id**. I tipi di dati salvabili sono molteplici (stringhe, interi a 32 o 64 bit, date, booleani, oggetti BSON, array BSON, null, ecc.).

```
1.  {
2.      "_id": 1,
3.      "name": "Claudio S.",
4.      "surname": "Di Mauro",
5.      "address": {
6.          "city": "Naples",
7.          "country": "Italy"
8.      },
9.      "hobbies": ["Photography", "Gym", "Technology", "Football"]
10. }
```

Snippet 1 – Esempio BSON

[8] **MongoDB** è stato sviluppato a partire dal 2007 e diventato open-source nel 2009, con licenza AGPL. È stato considerato production ready con la versione 1.4 di marzo 2010 e rilasciato in versione stabile nel febbraio 2015, con la versione 3.0 [10].

[9] **JSON**, *JavaScript Object Notation* è un formato di file utilizzato nell'interscambio di dati tra applicazioni client-server. Basato sul JavaScript, ma totalmente indipendente da esso, è compatibile con la maggior parte dei linguaggi di programmazione attualmente in uso.

La struttura "gerarchica" di Mongo prevede che all'interno di un database siano presenti una o più *collections* che a loro volta contengono i documenti. Per fare un confronto con i RDBMS, possiamo dire che una collection è equiparabile ad una tabella e un documento è assimilabile ad un record di questa tabella; i campi del documento rappresentano le colonne della tabella.

Essendo dunque basato su documenti, MongoDB risulta essere molto più flessibile rispetto ai classici DB relazionali, in più, cosa fondamentale, permette di indicizzare qualsiasi campo del documento per ottimizzare la ricerca.

Mongo fornisce un ricco e potente linguaggio di query per la creazione, ricerca, modifica e cancellazione dei documenti (operazioni **CRUD**[10]).

Si è dunque potuto già in parte capire i motivi che hanno spinto ad utilizzare MongoDB come scelta progettuale per questo lavoro di tirocinio. È necessario, però, evidenziare altre due caratteristiche fondamentali che hanno portato a questa scelta: la prima è sicuramente il fatto che Mongo sia un database **scheme-less**, ovvero non necessita di uno schema standardizzato per tutti i documenti, ma ogni documento può contenere informazioni differenti dagli altri; una seconda feature interessante è la possibilità, tramite la specifica **GridFS,** di salvare file che eccedano la dimensione massima dei documenti BSON, il cui limite massimo è 16MB.

GridFS è una sorta di *filesystem* che permette il salvataggio e il recupero di file come immagini, pdf, audio o video: tali dati vengono archiviati sfruttando le collection di MongoDB.

GridFS divide ogni file in blocchi detti **chunks** e memorizza ogni blocco di dati in un documento separato, ciascuno dei quali della dimensione massima di 255kB. Di default GridFS utilizza due collections, **fs.files** e **fs.chunks** per memorizzare rispettivamente i metadati dei file e i blocchi che li compongono. Ogni blocco è identificato dal proprio campo id univoco.

[10] **CRUD**, *Create Read Update Delete*, sono i principali metodi di accesso alle informazioni sui database NoSQL.

2.2.5 Robo3T

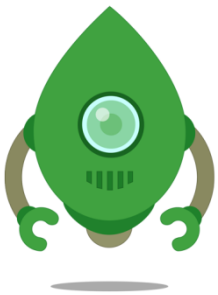

Fig. 8 - Logo Robo3T

Robo3T, precedentemente sviluppato sotto il nome di **Robomongo**, non è altro che un'interfaccia grafica nata per facilitare l'utilizzo di MongoDB, che altrimenti dovrebbe essere usato da *terminale*[11] (o *prompt dei comandi*, se si è in ambiente Windows) tramite linea di comando.

Esistono vari tools che permettono di visualizzare e gestire graficamente questo database, tuttavia la scelta è ricaduta su Robo3T per vari motivi. In primis, è di semplice utilizzo, una volta stabilita la connessione al database inserendo indirizzo e credenziali, è immediato comprenderne il funzionamento: vengono mostrati a video tutti i database presenti, con annesse collections e su di essi è possibile lanciare query per la creazione e la cancellazione di database o collections, per la ricerca o per il "drop" di un documento.

Con Robo3T la riga di comando non viene esclusa: questo software implementa una shell che si interfaccia attivamente con Mongo e che permette il lancio di query direttamente scrivendo il codice.

Altra caratteristica fondamentale che ha portato Robo3T a far parte di questo lavoro di tirocinio è quella di essere un progetto open-source e soprattutto gratuito, con alle spalle una vasta community a supportarlo.

[11] **Terminale**, in ambienti *Unix-like* (come Linux o MacOS), è un'interfaccia testuale che permette di interagire con il sistema operativo tramite riga di comando. Nei sistemi come MacOS l'app terminale è un emulatore che però mantiene tutte le caratteristiche di una vera *shell* Unix. Un qualcosa di simile si trova anche nella famiglia di s.o. Windows NT, sotto il nome di **prompt dei comandi**.

2.2.6 Postman

Postman è un comodo e utile strumento, strutturato come piattaforma collaborativa che permette di sviluppare, testare e distribuire le API REST.

Fig. 9 - Logo Postman

Postman consente di effettuare delle chiamate API senza dover mettere mano al codice dell'applicazione, fornendo un'interfaccia grafica e molto user-friendly che facilita l'invocazione dei vari metodi (GET, PUT, POST, DELETE, OPTION, ecc.).

È un software molto ben documentato e supportato ed è disponibile sia come addon su browser Chrome, sia come applicazione standalone. Tramite l'interfaccia è possibile specificare il tipo del metodo da chiamare e l'API specifica, oltre alla possibilità di inserire nell'invocazione parametri, header, body e script.

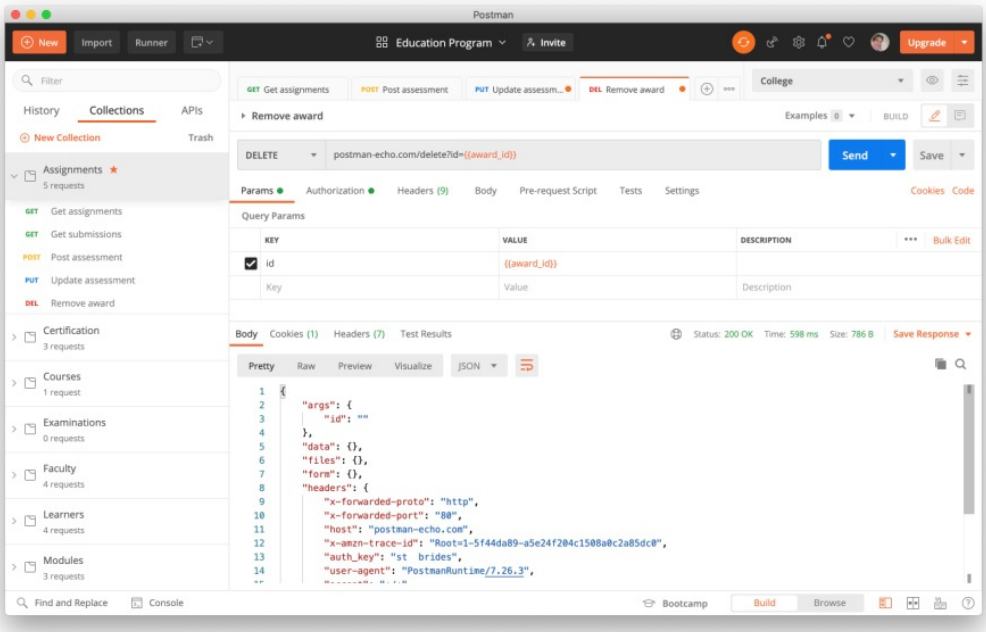

Fig. 10 - Interfaccia Postman

2.2.7 Swagger

Fig. 11 - Logo Swagger

Swagger [11] è un linguaggio di descrizione per la documentazione delle API RESTful espresse utilizzando JSON. Viene utilizzato insieme ad una serie di strumenti software open-source per progettare, documentare e utilizzare i servizi Web RESTful.

Nel lavoro di tirocinio, oggetto di questa tesi, Swagger viene utilizzato in virtù del fatto che consente di generare in maniera automatizzata la documentazione delle API e i vari test-case.

2.2.8 Docker

Docker [12] è una piattaforma open-source che permette l'esecuzione di applicazioni e facilita il processo di deployment delle stesse, aumentando energicamente anche la velocità di testing.

Fig. 12 - Logo Docker

Le applicazioni costruite su Docker sono "impacchettate" insieme a tutte le dipendenze necessarie al funzionamento, in un supporto chiamato **container**. Le tecnologie basate su container esistono da oltre 10 anni, tuttavia solo con l'avvento di Docker, la cui prima release risale al 2013, si stanno accingendo a diventare uno standard per lo sviluppo software.

Il concetto di container è molto vicino a quello di **macchina virtuale**, ma vedremo la sottile differenza che c'è tra le due cose. Quello che Docker si prefigge di fare

è rendere disponibili le applicazioni su qualsiasi ambiente, mantenendo il controllo sul codice eseguito, per il quale Docker fornisce una modalità standard basata su un "sistema operativo per container". Così come una macchina virtuale virtualizza i server hardware, così i container virtualizzano il sistema operativo di un server operando in maniera isolata dal **kernel** del sistema operativo ospitante.

Il container fa sì che il singolo processo rimanga limitato all'interno di una serie di librerie, di un *filesystem* e di una serie di risorse che si definiscono a priori.

La differenza sostanziale con la macchina virtuale è che non si ha un device driver per la scheda di rete, perché non si ha fisicamente a disposizione un disco o un file-system. Un'altra differenza considerevole è che una macchina virtuale è un oggetto "pesante" anche decine di GB, mentre il container è un'entità molto più snella e adeguata alle specifiche esigenze.

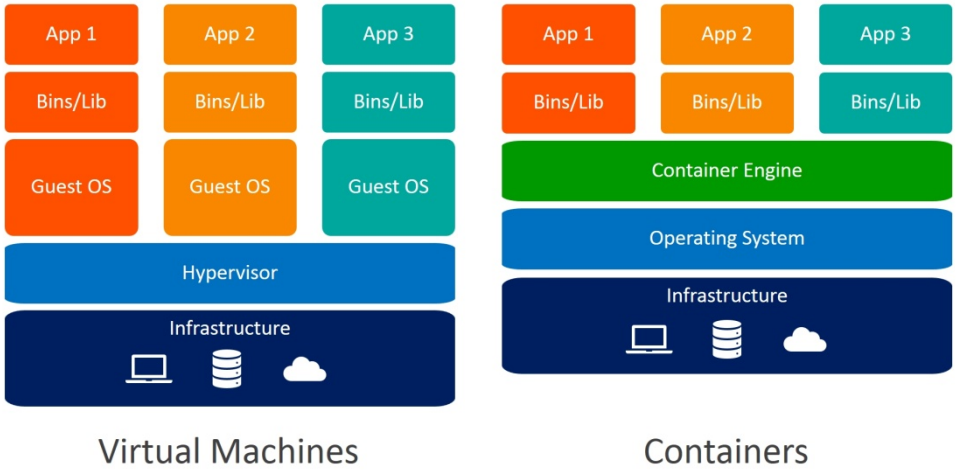

Fig. 13 - Virtual Machines vs Containers

La Virtual Machine è gestita da un **hypervisor** installato su un host fisico. Ogni app gira su un Guest OS differente e quindi ci saranno tanti Guest OS quante saranno le applicazioni in esecuzione.

Un container gira in generale su un **Container Engine**, che nel caso specifico di Docker prende il nome di **Docker Engine** ed è installato sull'host fisico. Il Docker Engine è unico per tutti i container in esecuzione.

Come già asserito, Docker si sta solidamente affermando come uno standard di sviluppo e pertanto la community che lo supporta è vastissima. Gli sviluppatori del progetto hanno messo a disposizione il **Docker Hub**, il più grande e completo repository di

immagini che possono girare su containers. Tutti i principali software sono stati "dockerizzati" come immagini e resi disponibili sull'Hub.

In virtù di ciò, la scelta di utilizzare Docker come strumento fondamentale nel lavoro di tirocinio si è resa quasi indispensabile, potendolo sfruttare nell'utilizzo di MongoDB ed eventualmente per effettuare un deploy futuro del lavoro stesso.

2.2.9 Git e GitHub

Git [13] è un progetto rilaciato a partire dal 2005 e sviluppato da Linus Torvalds (l'inventore di Linux). Tale progetto rappresenta il più utilizzato software di controllo di versione distribuito, ossia una particolare tipologia di controllo di versione che permette di tenere traccia delle modifiche effettuate e delle versioni apportate sul codice sorgente di un software su cui lavorano più sviluppatori.

Fig. 14 - Logo Git

Con tale sistema, [14] gli sviluppatori possono collaborare individualmente o parallelamente al progetto pur non essendo in diretto contatto con gli altri, creando un proprio ramo di lavoro (**branch**), registrando le modifiche di lavoro (**commit**) ed inseguito condividendo (**push**) o unendo il tutto a quelle degli altri (**merge**). Il tutto può avvenire senza il supporto di un server centralizzato.

Fig. 15 - Logo GitHub

GitHub [15] è un servizio di hosting per progetti software ed è una diretta implementazione dello strumento Git. Il sito permette agli sviluppatori di caricare il codice

sorgente dei propri programmi e condividerlo con tutta la community, rendendolo eventualmente scaricabile.

Con la sua evoluzione temporale, GitHub è diventato un vero e proprio strumento per l'agevolazione del workflow dello sviluppo software, tanto da mettere a disposizione delle bacheche condivisibili su cui poter annotare informazioni, stilare delle *to do list*, valutare i lavori in corso e i progressi fatti nel corso della progettazione.

Essendo l'oggetto di questo lavoro di tesi un progetto di tirocinio sviluppato insieme ad un altro studente, l'utilizzo di GitHub si è dimostrato fondamentale ed essenziale per poter portare avanti il team di sviluppo in maniera tale che ogni attore fosse sempre costantemente aggiornato sull'operato dell'altro e ad egli collegarsi.

CAPITOLO 3: LA SOLUZIONE PROPOSTA

Il terzo capitolo prevede l'esposizione dettagliata della soluzione proposta, con particolare attenzione su quelle che sono state le fasi di lavoro e come queste hanno portato a cambiamenti, anche radicali nel risultato finale.

Si parte da uno scraping "statico" fino ad arrivare ad uno "dinamico", passando per MongoDB come base dati di appoggio per i dati analizzati. Microservizi ed esposizione di REST APIs, insieme al Web Scraping, sono il fulcro centrale di tutto il progetto.

3.1 L'idea

3.1.1 Fase iniziale e approccio agli strumenti

La primissima fase del lavoro di tirocinio ha visto venir fuori quelle che erano le scelte da effettuare riguardo quelle che potessero essere le migliori soluzioni da adottare.

Come già esposto nel capitolo 2 del presente elaborato, ogni scelta e ogni tecnologia utilizzata è stata ponderata al fine di ottenere sia un buon risultato, sia di poter sfruttare le community di sviluppatori a supporto delle suddette.

Dopo aver scelto quelle che dovevano essere le tecnologie abilitanti per il progetto, almeno in una fase embrionale, si è iniziato a prendere confidenza con queste, dato che erano strumenti mai utilizzati prima. Si è dunque dovuto fare un focus su ognuna di esse per poterle padroneggiare e quindi maneggiare in maniera adatta.

A seguito di un confronto con i tutor aziendali, l'idea di base è stata quella di "lavorare per step", completando di volta in volta dei task e arrivando ad ottenere almeno quel requisito minimo da presentare, per poi arricchirlo con ulteriori features sempre più avanzate.

3.1.2 Lo scraping "statico"

Il processo di Web Scraping è, di base, un qualcosa che prevede la conoscenza del DOM delle pagine di cui si sta effettuando l'estrazione dati, poiché a partire da esso è possibile sfruttare i tag e gli attributi per ricavarne le informazioni.

La libreria JSoup, utilizzata per questo progetto, offre una importante quantità di metodi che permettono la manipolazione dei dati presenti nel DOM di una pagina HTML e grazie proprio a questa sua caratteristica è stato uno strumento che si è prestato davvero bene allo scopo: i primi test effettuati hanno previsto di prendere determinate pagine e ottenere alcuni dati di interesse, andando a "costruire" del codice Java che si basasse interamente su quelle parti di interesse della risorsa online.

Si allega di seguito un esempio di scraping, in cui si è andati ad estrapolare le informazioni relative alla classifica della Serie A.

```
1.  final String url = "http://www.legaseriea.it/it/serie-a/classifica";
2.
3.  Document doc = Jsoup.connect(url).timeout(10000).get;
4.
5.  for(Element row : doc.select("div#classifiche tbody tr")) {
6.     String position = row.select("td span.pos").text();
7.     String team = row.select("td img").attr("title");
8.     String score  = row.select("td.blue").text();
9.
10.    System.out.println(position + " | " + team + " - " + score + "pti.");
11. }
```

Snippet 2 – Esempio Web Scraping

Quello che questo semplice codice fa è connettersi all'url definito dalla prima stringa (dichiarata come **`final`**, dato che ci si aspetta che questa non cambi), mediante il metodo **`connect`** invocato direttamente sulla classe **`Jsoup`**. Il metodo **`timeout`** definisce il tempo massimo (espresso in ms) entro cui mantenere aperta la connessione: se non specificato, di default la connessione resterà aperta per 30s, mentre se come parametro gli si passa 0, la connessione resterà aperta per un tempo infinito.

Il messaggio **`select()`**, inviato alla variabile doc di tipo **`Document`** (**`org.jsoup.nodes.Document`**), restituisce l'elemento ottenuto leggendo i tag del DOM che gli si è passato come stringa. Un ciclo **`for each`** su tutti gli elementi rinvenuti sotto il tag **`tr`** nella gerarchia **`div#classifiche tbody`**, permette di ottenere tutte le informazioni presenti.

La **`System.out.println()`**, infine, produrrà una stampa su console del tipo:

```
1 | INTER - 50pti.
2 | MILAN - 49pti.
3 | ROMA - 43pti.
4 | JUVENTUS - 42pti.
5 | NAPOLI - 40pti.
…
```

Dovrebbe a questo punto risultare chiaro il fatto che una siffatta procedura sia effettivamente del tutto statica, essendo note a priori la sorgente Web su cui lavorare, le informazioni da estrarre e i tag da utilizzare per farlo.

3.1.3 L'introduzione del database

Avere dei dati estratti da una sorgente web è sicuramente qualcosa di utile, specie in determinate circostanze aziendali. Tuttavia garantire la persistenza dei dati estratti è un qualcosa di ancora più importante dato che, senza la persistenza questi dati sarebbero leggibili solo nel momento in cui si effettua lo scraping e non anche in momenti futuri. L'utilizzo di una base di dati per il salvataggio (e il successivo recupero) di tali informazioni è risultata essere quindi indispensabile per il presente progetto.

Come si è già detto nel paragrafo 2.2.4, si è scelto di utilizzare come base di dati un particolare tipo di struttura: quella non relazionale. Nello specifico la scelta è ricaduta, com'è noto, su MongoDB, per il quale sono disponibili dei *driver Java* [16] che permettono di interfacciarsi al codice in maniera semplice e chiara.

Per questo tipo di applicazione, è parso essenziale discostarsi dai modelli relazionali dei classici database (si pensi ad esempio a MySQL), dato che un utente può scegliere di memorizzare una serie eterogenea di informazioni, ognuna con una struttura diversa a seconda della sorgente "raschiata". Un database NoSQL, quindi non relazionale come Mongo, permette di fare ciò, impostando ogni documento secondo un proprio schema (si ricordi che Mongo è scheme-less, ossia non ha bisogno di una struttura standard per i documenti).

In maniera statica, la connessione al database può avvenire in vari modi: uno dei più semplici da utilizzare è quello di sfruttare un'istanza **`MongoClient`** (**`com.mongodb.MongoClient`**), a cui si passa una **`MongoClienURI`** (**`com.mongodb.MongoClientURI`**) per instaurare la connessione al db:

```
1.  private final String dbURI = "mongodb://root:pippo@localhost:27017";
2.
3.  MongoClient mongoClient = new MongoClient(new MongoClientURI(dbURI));
4.
5.  MongoDatabase database = mongoClient.getDatabase("CrudDB");
6.  MongoCollection<Document> collection = database.getCollection("persone");
7.
8.  //Create
9.  org.bson.Document document = new Document()
10.                              .append("name","Claudio")
11.                              .append("surname","Di Mauro")
12.                              .append("age","27")
13.                              .append("gender","Male");
14. collection.insertOne(document);
15.
16. //Read
17. com.mongodb.client.FindIterable<Document> allDocuments = collection.find();
18. for(org.bson.Document doc : allDocuments) {
19.    System.out.println(doc.toJson(););
20. }
21.
22. //Update
23. collection.updateOne(
24.          eq("name", "Claudio"),
25.          combinse(set("name", "Claudio Salvatore"), set("age", "27 e mezzo"))
26.          );
27.
28. //Delete
29. collection.deleteMany(eq("name", "Claudio Salvatore"));
```

Snippet 3 – Esempio connessione a MongoDB

La URI[12] è passata al client sotto forma di stringa nel formato **`mongodb://user:password@nomedominio:port`**.

Con il metodo **`getDatabase()`** invocato sulla variabile **`mongoClient`** si "ottiene" la connessione al nome della specifica base di dati contenuta in Mongo e, qualora questa non esistesse, all'atto del salvataggio viene creata. Stesso discorso per il metodo **`getCollection()`** chiamato su **`database`**.

Le operazioni effettuate in seguito a quanto appena detto, sono le cosiddette *operazioni CRUD*, che si occupano rispettivamente di inserire (Create), leggere (Read), aggiornare (Update) e cancellare (Delete) le risorse dal database.

[12] **URI**, *Uniform Resource Identifier*, è una sequenza di caratteri che identifica in maniera univoca una risorsa su rete. Un esempio di URI è l'URL.

3.1.4 Il passaggio ai microservizi: il framework Spring e l'esposizione di REST APIs

Come anticipato nei capitoli precedenti, l'idea di base è stata quella di portare avanti questo progetto di Web Scraping sfruttando quella che è l'**architettura a microservizi**. Si è già spiegato come questa tipologia di architettura riesca a rendere il software molto più snello e mantenibile, andando a dividere le sue parti in tanti "servizi", ognuno indipendente dall'altro.

Un framework che in Java permette di fare ciò è **Spring** che, con le sue librerie fornisce innumerevoli funzionalità. Di per sé, Spring nasce per lo sviluppo di applicazioni Web e pertanto necessita di tutta una serie di configurazioni sui file XML e la presenza di un server http che sia in grado di farlo girare. **Spring Boot** è una naturale evoluzione (ed estensione) di questo framework che prende vita con l'intento di sollevare lo sviluppatore da tutte quelle che sono le configurazioni iniziali di Spring.

Un progetto Spring Boot prevede di inglobare al suo interno tutte le dipendenze necessarie al corretto sviluppo del software, nonché un Web Server basato su **Tomcat**[13]. Il metodo più semplice per avviare una *Spring Boot Application* è quello di connettersi al sito ufficiale, *https://start.spring.io*, selezionare la versione del linguaggio e del framework Spring Boot, lo strumento di gestione del progetto (nel caso del lavoro in oggetto si è usato **Apache Maven**) e le dipendenze necessarie per poi avviare il download di un progetto già pronto da importare nel proprio IDE di lavoro e scrivere tutto il codice del caso.

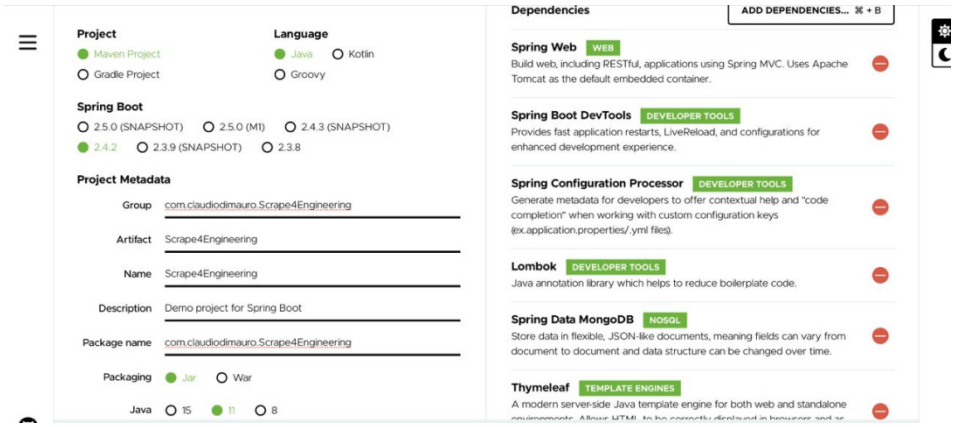

Fig. 16 - Interfaccia start.spring.io

[13] **Tomcat** è un server Web open-source sviluppato dalla *Apache Foundation*, che implementa al suo interno le specifiche che consentono lo sviluppo Web basato su JavaService Page (JSP) e Servlet.

Per il lavoro di Web Scraping Automation, le dipendenze inserite tramite la piattaforma spring.io, non sono state sufficienti e, all'apertura del progetto scaricato da spring.io si è dovuto aggiungere altro codice (con altre dipendenze) all'interno del file **POM.xml**. Per dipendenze si intendono tutte quelle librerie che contribuiscono al funzionamento di un progetto; in particolare nel caso in esame le dipendenze utilizzate sono state le seguenti:

- **Spring Web**: include Apache Tomcat e le funzionalità per le RESTful API;
- **Spring Boot DevTools**: include strumenti per migliorare e agevolare l'esperienza di sviluppo;
- **Spring Configuration Processor**: genera i file per gestire i metadata del progetto, come ad esempio il file *application.properties*, in cui è possibile specificare le caratteristiche che deve avere l'applicazione (e.g.: la porta del server su cui lavorare, il DB a cui connettersi, il caching del web server, ecc.);
- **Spring Data MongoDB**: include i driver e le librerie da utilizzare per interfacciare l'applicazione al database Mongo;
- **Thymeleaf**: è un template engine HTML per far sì che sul server possano girare correttamente le pagine HTML da usare come template per il progetto;
- **Lombok**: contiene tutta una serie di annotazioni che permettono di snellire il codice in maniera semplice e veloce (e.g.: `@getter` e `@setter` costituiscono un modo estremamente rapido per generare automaticamente – e senza che nel codice della classe ce ne sia visibilità – tutti i metodi accessóri e mutatóri);
- **JSoup**: insieme a Spring Web costituisce il cuore di tutto il lavoro, dato che in questa libreria sono contenuti i metodi per effettuare lo scraping vero e proprio;
- **Springfox-Swagger2**: contiene tutti i metodi e le annotazioni necessarie alla documentazione delle APIs;
- **Springfox-Swagger-ui**: implementa le funzionalità servlet per creare una interfaccia grafica in formato HTML che, quando visitata mediante l'indirizzo *https://nomeserver:port/swagger-ui.html*, restituisce una struttura visivamente semplice da comprendere per la lettura della documentazione e il testing delle stesse APIs.

Tutte quelle dipendenze che non sono state recuperate in automatico alla creazione del progetto Spring Boot, sono poi state ricercate tra i repository Maven (*https://mvnrepository.com/)* e inserite manualmente nel file POM.

Portare avanti un progetto Spring Boot, significa adottare automaticamente uno dei più noti *design pattern* in uso, ossia il **Model-View-Controller** (**MVC**) [17] che prevede di suddividere il software in tre macro elementi ognuno dei quali con un compito ben specifico:

- Il **model** deve incapsulare al suo interno dati e funzionalità e mantenersi indipendente dal comportamento interattivo del sistema;
- La **view**, come suggerisce il nome stesso, è quella componente che mostra all'utente finale le informazioni; spesso è presentata sotto forma di interfaccia grafica;
- Il **controller** è quella componente che si occupa di operare sui dati e sfruttare le funzionalità dei modelli; interagisce in maniera diretta con l'utente.

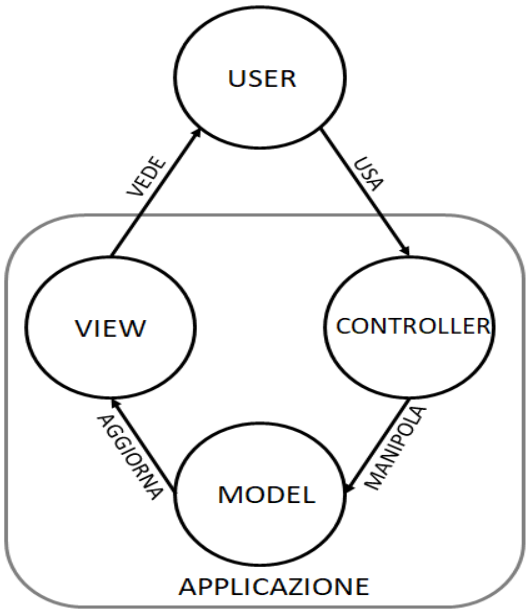

Fig. 17 - Pattern architetturale Model-View-Controller

In ottemperanza a tale pattern, anche il lavoro di tirocinio è stato sviluppato in questo modo, andando a suddividere il tutto in diversi packages, ognuno dei quali contenitore di uno o più elementi:

`com.claudiodimauro`

`com.claudiodimauro.api.controllers`

```
com.claudiodimauro.api.models
com.claudiodimauro.api.repositories
com.claudiodimauro.api.resources
com.claudiodimauro.api.services
com.claudiodimauro.configurators.
```

Questa struttura ad albero agevola lo sviluppatore nel tenere tutto ordinato e a rendere "più visibile" l'architettura MVC. Nel package base è presente la classe main, quella che permette l'avvio del programma e che identifica la Spring Boot Application mediante una omonima annotazione (**`@SpringBootAnnotation`**); il package **`configurators`** contiene al suo interno la classe che effettua la configurazione di Swagger2. Il contenuto di **`models`** e di **`controllers`** è facilmente intuibile per quanto detto sul MVC.

Vale forse la pena fare invece un focus sul contenuto di **`repositories`**, **`services`** e **`sources`**.

`Repositories` e **`services`** sono due package strettamente connessi all'utilizzo delle dipendenze MongoDB nel progetto, in particolare: **`repositories`** contiene delle interfacce che estendono la **`MongoRepository`** e che sono utili alla manipolazione dei dati da passare al database; **`services`**, invece contiene i "servizi" che permettono la manipolazione effettiva dei dati, andando di fatto ad implementare quelle che sono le operazioni CRUD. Un service è identificato mediante l'annotazione **`@service`** posta subito prima della dichiarazione della classe.

`Sources` contiene quelle classi di appoggio che sono utilizzate per la manipolazione degli elementi; i metodi contenuti in questo package vengono utilizzati in maniera ampia all'interno dei controllers per effettuare le operazioni richieste.

All'interno dei controllers sono "esposte" le API che permettono di effettuare le operazioni richieste. In particolare le API rappresentano delle richieste HTTP che quando eseguite permettono di lanciare il metodo a cui sono associate.

Per il software in esame, si è scelto di sviluppare diverse REST APIs – alcune delle quali verranno approfondite in seguito – che si incentrassero sulla creazione, la cancellazione, l'aggiornamento e lo scraping di quelle informazioni non note a priori e che sono state definite "entity", intendendole come entità generiche che altro non sono che le informazioni su cui viene effettuato lo scraping.

Un esempio di API, direttamente prelevato dal codice sorgente del programma sviluppato, può essere il seguente snippet:

```
@GetMapping("/getEntity/{id}")
    public Entity getById(@PathVariable("id") String id) {
        return entityService.getById(id)
                .orElse(null);
    }
```

Snippet 4 – Esempio GetMapping

Con l'annotazione **`@GetMapping`** si sta andando ad effettuare una vera e propria mappatura sul metodo GET della funzione **`getById()`**, dove **`{id}`** rappresenta un parametro da passare al metodo GET per poter effettuare una corretta richiesta REST. Il tutto funziona, a patto che venga "annotata" la classe ospitante il metodo come una **`RestController`**:

```
@RestController
@RequestMapping("/api")
public class EntityController {...}
```

Snippet 5 – Il RestController

Essendo stato fatta una **`RequestMapping`** anche su tutto il controller, cioè su tutta la classe, sarà possibile effettuare la richiesta REST per chiamare il metodo **`getById()`** andando a digitare nel browser o in un qualsiasi altro client HTTP utile per la circostanza (in seguito vedremo il tutto con Postman e con Swagger2) la stringa:

https://nomeserver:porta/api/getEntity/1234

dove 1234 può essere considerato come l'id dell'entità da cercare.

3.2 Lo sviluppo definitivo

3.2.1 Lo scraping "dinamico"

Fin dall'inizio l'idea è stata quella di riuscire a realizzare un software di scraping che potesse adattarsi a qualsiasi sorgente, senza dover apportare modifiche nel codice sorgente.

Effettuare un'analisi generica per estrarre informazioni dal Web non è sicuramente un qualcosa di facile, immediato o, talvolta, fattibile, dato che ogni pagina Web

sviluppa un DOM che è differente dalle altre e che richiede una struttura precisa nel momento in cui si va ad utilizzare JSoup per lo scraping.

Riuscire, allora, a generalizzare il tutto è stato un lavoro al quanto arduo e si è giunti alla conclusione di poter sviluppare un qualcosa che, però, richiedesse comunque una conoscenza del DOM da parte dell'utente finale.

La prima cosa fatta è stata andare ad astrarre il concetto di entità, intesa come ciò che l'utente vuole estrapolare, dopodiché si è definita una struttura che potesse interfacciarsi con l'entità ed essere manipolata dall'utente in base alla pagina da "raschiare".

3.2.2 Pattern per lo scraping semi-dinamico

La soluzione pensata per poter portare avanti un lavoro che si presentasse come dinamico (o semi-dinamico), è stata quella di ideare un pattern che, sempre tramite delle REST APIs potesse ricevere dall'utente le informazioni necessarie a lavorare sul DOM, ossia quei tag e quegli attributi HTML che permettessero a JSoup di lavorare correttamente.

```
1.  @Getter
2.  @Setter
3.  @Document(collection = "patterns")
4.  public class Pattern {
5.      @Id
6.      private String patternName;
7.      private String url;
8.      private String tagForBody;
9.      private String entityId;
10.     private String entityPath;
11.     private String attrForEntityId;
12.     private Boolean hasPrescraping = false;
13.     private String tagForPrescraping;
14.     private Boolean haveToExplore = false;
15.     private List<PatternObject> patternObjects;
16.     private List<PatternObject> innerPatternObjects;
17.     private String tagForInnerBody;
18.     private Boolean hasAttachments = false;
19.     private List<AttachmentObject> attachmentObjects;
20.     private Boolean hasInnerAttachments = false;
21.     private List<AttachmentObject> attachmentInnerObjects;
22.
23.     public Pattern() {
24.         this.patternObjects = new ArrayList<>();
25.         this.innerPatternObjects = new ArrayList<>();
26.         this.attachmentObjects = new ArrayList<>();
27.         this.attachmentInnerObjects = new ArrayList<>();
28.     }
29. }
```

Snippet 6 – La classe Pattern

Con le annotazioni Lombok **`@Getter`** e **`@Setter`** si sta dicendo al compilatore di andare automaticamente a creare i metodi di Get e Set per tutti gli attributi della classe; con **`@Document(collection = "patterns")`** si è definito il nome della collection nel database Mongo che deve contenere tutti i pattern creati; l'annotazione **`@Id`** indica quale tra questi attributi deve essere la chiave primaria del database: se non specificata, Mongo assegna di default un id al documento.

Per una documentazione più dettagliata si rimanda a quella ufficiale del progetto, tuttavia è bene analizzare alcuni attributi e il loro scopo.

- **`patternName`** rappresenta l'identificativo (id) con il quale si rende univoco il pattern sul database; è una stringa contenente il nome che si vuole assegnare allo stesso.
- **`url`** rappresenta il link alla risorsa online che si vuole analizzare e da cui si vogliono estrapolare i dati.
- **`tagForBody`** è il tag principale che nel DOM va ad isolare la parte contenente l'informazione da estrapolare.
- Solitamente in una pagina ci sono più informazioni da estrapolare e secondo lo schema stabilito, ogni informazione è un'entità. Assegnare un identificativo "locale" ad ogni entità è una buona norma per tenerne traccia: **`attrForEntityId`** e **`entityId`** si occupano di fare ciò, rispettivamente prendendo l'attributo passato dall'utente (solitamente di tipo **`href`**) tramite **`attrForEntityId`**, fare l'estrapolazione dell'informazione richiesta e poi assegnarla a **`entityId`**, che la mantiene "salvata" fino a passarla a quello che sarà l'oggetto entità di cui si parlerà nel prossimo paragrafo.
- **`hasPrescraping`** è una variabile booleana atta ad indicare se c'è bisogno di effettuare uno scraping iniziale per capire se ci sono più pagine da analizzare o la pagina data è unica.
- **`haveToExplore`** è un attributo strettamente collegato a **`tagForInnerBody`**: essi permettono di capire se c'è bisogno di analizzare un contenuto "interno" (ossia cliccando su un link per accedere al contenuto vero e proprio – e.g.: si pensi a quando si è su un sito di articoli, dove c'è una lista di contenuti ma per leggere l'articolo di interesse si deve cliccare sull'anteprima) e tramite quale tag farlo. Le pagine "interne" sono state definite come "inner".

Di fondamentale importanza sono le classi **`PatternObject`** e **`AttachmentObject`**, le quali hanno innanzitutto il compito di snellire il codice di **`Pattern`**, ma hanno anche il compito di andare ad effettuare una ulteriore generalizzazione sui contenuti da estrapolare, dato che queste definiscono degli attributi che contribuiscono al retrieving delle informazioni. In particolare **`AttachmentObject`** è una classe che si occupa di gestire i tag e gli attributi HTML per far sì che si possano scaricare gli allegati (video, immagini, documenti, …) ed entra in gioco solo nel momento in cui l'utente definisce un pattern in cui la variabile **`hasAttachments`** (o **`hasInnerAttachments`**, nel caso in cui ci fossero pagine interne – o anche solo allegati interni – da analizzare) fosse settata a **`true`**.

Le liste di "sotto pattern" definite tra gli attributi della classe servono a mantenere al loro interno tutta una serie di entità, entità "inner", allegati, ecc., dato che difficilmente da una pagina si vorrà estrapolare una sola informazione; nella stragrande maggioranza dei casi ciò che si andrà a fare sarà una iterazione per la lettura del DOM (nei tag specificati) nella sua totalità in modo da prelevare tutte le entità e tutti gli allegati presenti.

```
1.  @NoArgsConstructor
2.  @Getter
3.  @Setter
4.  public class PatternObject {
5.      private String elementToScrape;
6.      private String tagForElementToScrape;
7.      private Boolean methodForElementToScrape;
8.      private @Nullable String attrForElementToScrape;
9.  }
```

Snippet 7 – La classe PatternObject

La variabile booleana **`methodForElementToScrape`** serve a specificare quale metodo JSoup va utilizzato per effettuare lo scraping. Di base, quando alle funzioni di "select" JSoup si passano dei tag HTML esistono due possibilità per effettuare il retrieving dell'informazione che essi racchiudono: si può ottenere il testo racchiuso nei tag utilizzando il metodo **`text()`**, oppure si può ottenere il testo presente all'interno di un attributo del tag usando **`attr(String s)`**, la cui stringa da passare rappresenta il nome dell'attributo. Entrambi questi metodi ritornano un tipo String. Si è strutturato il codice in modo tale che se **`methodForElementToScrape==true`**, allora il metodo da usare è **`text()`**, altrimenti è da usare **`attr(String s)`**.

Contestualmente a questo attributo si utilizza **attrForElementToScrape** che, come suggerisce il nome, rappresenta l'attributo di cui si vuole ottenere il testo. È **@Nullable** poiché può essere settato a **null** nel caso in cui si ha **methodForElementToScrape==true**, dato che in questo caso non è richiesto l'utilizzo del metodo **attr(String s)** ma del metodo **text()**.

Si noti che **@NoArgsConstructor** è un'altra delle annotazioni appartenenti alla libreria Lombok e ha il compito di dichiarare in maniera rapida il costruttore senza argomenti per la classe su cui effettua l'annotazione.

elementToScrape e **tagForElementToScrape** sono due variabili che rispettivamente servono a dare un titolo a ciò che si sta estraendo e il tag tramite cui deve essere fatto.

```
1.  @NoArgsConstructor
2.  @Getter
3.  @Setter
4.  public class AttachmentObject {
5.      private String tagForElementToScrape;
6.      private String attrForElementToScrape;
7.  }
```

Snippet 8 – La classe AttachmentObject

Come si può osservare, **AttachmentObject** è una classe molto simile a **PatternObject** appena vista e, di fatti, lo scopo è pressoché lo stesso: **AttachmentObject** definisce delle variabili su cui appoggiarsi per far passare all'utente i tag e gli attributi necessari allo scraping di immagini, video, pdf, ecc.

In questo caso, non ha avuto senso dichiarare variabili tipo **elementToScrape** e **methodForElementToScrape**, questo perché di base è sempre possibile ricavare un titolo a partire dal nome della risorsa che si va a scaricare e il metodo per ottenere il link in una risorsa è sempre contenuto in un tag il cui attributo sarà **href**; ciò significa che la funzione utilizzata per l'ottenimento dell'indirizzo sarà sempre **attr(String s)**.

3.2.3 Definizione delle entità e sviluppo centrale

```
1.  @Document(collection = "entities")
2.  public class Entity {
3.
4.      @Id
5.      private String id;
6.      private String entityId;
7.      private String basePath;
8.      private String path;
9.      private Date lastScraping;
10.     private DBObject content = new BasicDBObject();
11.     private List<String> attachmentIds = new ArrayList<>();
12.     private DBObject entityObject = new BasicDBObject();
13.
14.     public void setEntityObject(String key, String value) {
15.         this.entityObject.put(key, value);
16.     }
17.     public void setContent(String key, String value) {
18.         this.content.put(key, value);
19.     }
20.     public void setAttachmentIds(String value) {
21.         this.attachmentIds.add(value);
22.     }
```

Snippet 9 – La classe Entity

Entity è la classe che definisce la generica entità "raschiata" da una pagina Web. Si parla di entità generica dato che non è noto a priori quello che l'utente deciderà di estrarre da un documento online: possono essere estratti i dati di una tabella, un articolo, una stringa semplice, e così via.

- In questa classe è definito un **`id`**, che però è lasciato essere assegnato da MongoDB; è tuttavia dichiarato in quanto è necessario in fase di update di una entità.
- Con **`entityId`** si definisce, invece, un id "locale" che potrebbe eventualmente anche essere duplicato sulla base dati.
- **`content`** è una variabile di tipo **`DBObject`** e rappresenta l'effettivo contenuto di interesse sull'entità: è dichiarata di questo tipo sempre in virtù dell'eterogeneità dei dati estratti, a ragione del fatto che ogni contenuto può essere diverso nella sua struttura da entità a entità.
- **`attachmentIds`** è una lista di stringhe, implementata come un **`ArrayList`**, di tutti gli id che rappresenteranno gli allegati di un'estrazione dati; serve essenzialmente a garantire un collegamento tra la entity e i suoi allegati – essendo questi posti in una propria collection.
- **`entityObject`** è utilizzata per settare il titolo e il contenuto dell'elemento che si sta andando a trattare.

Come si evince dall'annotazione **`@Document`**, ogni entità trovata verrà salvata in una collection del database che andrà sotto il nome di "entities".

Lo sviluppo centrale di tutto il software è fondamentalmente affidato alle due classi del package **`resources`**, ovvero **`ScrapeByPattern`** e **`ScrapeByNewPattern`**, le quali offrono l'implementazione dei metodi per effettuare la scansione delle pagine e ricercare così i contenuti di interesse da una pagina singola, da più pagine (a seguito del prescraping), degli allegati, e via discorrendo.

Degli oggetti di queste classi sono poi istanziati nei controllers e su di essi vengono invocati i messaggi da utilizzare per effettuare le operazioni suddette. In particolare, con **`ScrapeByPattern`** si ottiene tutta un'interfaccia tra utente e software per poter effettuare lo scraping mediante dei pattern già presenti sulla base dati; viceversa, con **`ScrapeByNewPatter`** è possibile andare a creare un nuovo pattern, sfruttarlo per lo scraping e infine ritrovarlo – a seguito di un salvataggio trasparente all'utente – sul DB.

3.2.4 La manipolazione dei dati

La manipolazione dei dati è stato un processo fondamentale al fine di individuare e salvare in maniera corretta i dati di interesse sul database.

Avendo utilizzato Mongo, che – come si è detto – è scheme-less, si è potuto mantenere un certo grado di libertà nella scelta delle informazioni da salvare. In generale quello che si è fatto è stato dividere la base dati in quattro collections:

1. **entities**, all'interno della quale è effettuato lo storage delle entità con gli annessi campi di interesse;
2. **patterns** è il contenitore di tutti i patterns creati dall'utente. Ogni pattern mantiene al proprio interno tutta una serie di elementi che, utilizzati correttamente, permettono lo scraping
3. **fs.files** e **fs.chunks** sono le collections generate dalla feature GridFS di Mongo, all'interno delle quali sono salvati rispettivamente i dati relativi agli allegati (nome, tipo, metadata, …) e i chunks in cui questi sono suddivisi.

Sarebbe dispendioso elencare uno ad uno tutti i campi delle collections di cui sopra, per cui a titolo di esempio si riportano solo i campi della collection entities:

- **_id**: generato da Mongo;
- **entityId**: è un id "locale", che identifica la risorsa in modo più diretto. Può essere duplicato;
- **basePath**: è l'indirizzo base del sito su cui si effettua l'estrazione dati (e.g.: www.nomesito.com);
- **path**: è il link alla risorsa, escluso il basePath (e.g.: /risorsa.html);
- **lastScraping**: è la data dell'ultima volta in cui la risorsa è stata cercata e ottenuta;
- **content**: è un object (una sorta di sotto-documento) che contiene le informazioni essenziali che si vuole ottenere;
- **attachmentIds**: è un array di stringhe che contiene dei riferimenti agli allegati;
- **entityObject**: contiene delle informazioni supplementari (come ad esempio il titolo di un articolo, la data dell'ultimo aggiornamento dello stesso, ecc.).

3.2.5 La "view": swagger-Ui e i Web controllers

Nei paragrafi precedenti si è evidenziato, anche nella suddivisione del progetto in packages, di come l'architettura del servizio fosse di tipo MVC. Si è, inoltre, già esplicitato quale classe fungesse da model e quale da controller, tuttavia non si è ancora specificato qual è quella entità facente funzione di view.

Solitamente la view è assimilabile ad una interfaccia grafica che consente all'utente di interagire con il software; un tipico esempio potrebbe essere una applicazione mobile, dove l'interfaccia finale mostrata all'utilizzatore rappresenta la view, la quale ha anche il compito di rendere trasparente agli users le implementazioni e le operazioni di fondo che servono ad eseguire determinate azioni.

Nel caso del progetto in esame, la view esiste anche se non è stata "manualmente scritta": stiamo parlando della Swagger-Ui, ossia l'interfaccia grafica che permette di leggere la documentazione e di testare/usare le API create.

Per questo tirocinio non è stata prevista (almeno in questa fase) una interfaccia grafica, per cui il ricorso alla Swagger-Ui è risultato ottimale. Grazie alle librerie Springfox-Swagger utilizzate, il programmatore è sollevato dal compito di scrivere del codice per ad hoc da usare come GUI (Graphic User Interface), ottenendo in maniera del tutto

automatica le voci che egli va a documentare. L'unico onere per il programmatore è andare a posizionare nei punti corretti delle annotazioni che permettano a Swagger di "capire" come e cosa documentare/prelevare/mostrare.

Di Swagger-Ui se ne parlerà in maniera più approfondita nel prossimo capitolo, in cui verranno mostrate le fasi di testing effettuate sul progetto.

È buona cosa fare, a questo punto, una breve digressione su quelli che sono i controllers "generici" che possono essere definiti usando Spring Boot. Essendo essenzialmente un framework mirato allo sviluppo delle Web Applications, con Spring è possibile gestire delle pagine HTML che andranno a fare da interfaccia in determinate circostanze, specificate dal programmatore.

Un esempio tipico di Web controller è la home page del server su cui ci si connette: è infatti possibile scrivere del codice HTML e codificare un Web controller in modo tale che riconosca quel file (e.g.: index.html) come pagina principale. Allo stesso modo è possibile definire una pagina per gli errori generati dal server, come potrebbe essere la classica pagina "ERRORE 404. PAGINA NON TROVATA".

CAPITOLO 4: TEST E CONCLUSIONI

Il testing e la "dockerizzazione" sono aspetti fondamentali del progetto presentato. In questo capitolo si parlerà proprio di questo, andando poi ad osservare quelli che potrebbero essere dei miglioramenti effettuabili e i possibili usi futuri del software sviluppato.

4.1 La fase di testing

4.1.1 Test con Postman e Swagger2

Lo sviluppo di questo progetto è stato portato avanti su diverse fasi, una delle quali ha previsto di effettuare tutte quelle prove che permettessero di padroneggiare gli strumenti di lavoro, le librerie da utilizzare e i framework scelti. Essendo, fondamentalmente, ognuno di questi pezzi sciolto dagli altri, le prove effettuate su ogni componente/strumento ha richiesto determinate operazioni. In particolare, quando si è cominciato a capire l'utilizzo delle REST APIs esposte con Spring Boot, si è fatto un forte uso del software Postman, tramite cui è stato possibile usare tutti i metodi HTTP per effettuare le operazioni richieste. Per ogni API si è creata una apposita sezione e la si è utilizzata tutte le volte che ce n'era bisogno.

Nel momento in cui lo sviluppo ha proseguito con la messa a punto di un qualcosa di definitivo, Postman è stato sostituito con Swagger2, quella libreria di cui abbiamo già parlato nei capitoli precedenti e che, oltre alla documentazione delle APIs, consente anche di essere usata come "view" per permettere all'utente del software di effettuare tutte le richieste HTTP che interessano.

Nel caso in esame, Swagger2 è stato utilizzato anche per la fase di testing del software, come si vedrà nel seguito del paragrafo. Avendo testato il tutto in locale, sulla macchina utilizzata per lo stesso sviluppo, non si è utilizzato un contenitore Docker per lavorare, ma si è sfruttato lo stesso IDE (NetBeans) per poter mandare in esecuzione il software e leggere i dati dalla console.

Si analizzeranno ora le fasi di testing essenziali all'ottenimento dei dati richiesti da un utente.

Il primo passo per testare/lavorare con il software è accedere all'interfaccia Web dello stesso, connettendosi tramite un browser all'indirizzo *https://localhost:8080/swagger-ui.html*, da cui viene mostrata la seguente schermata:

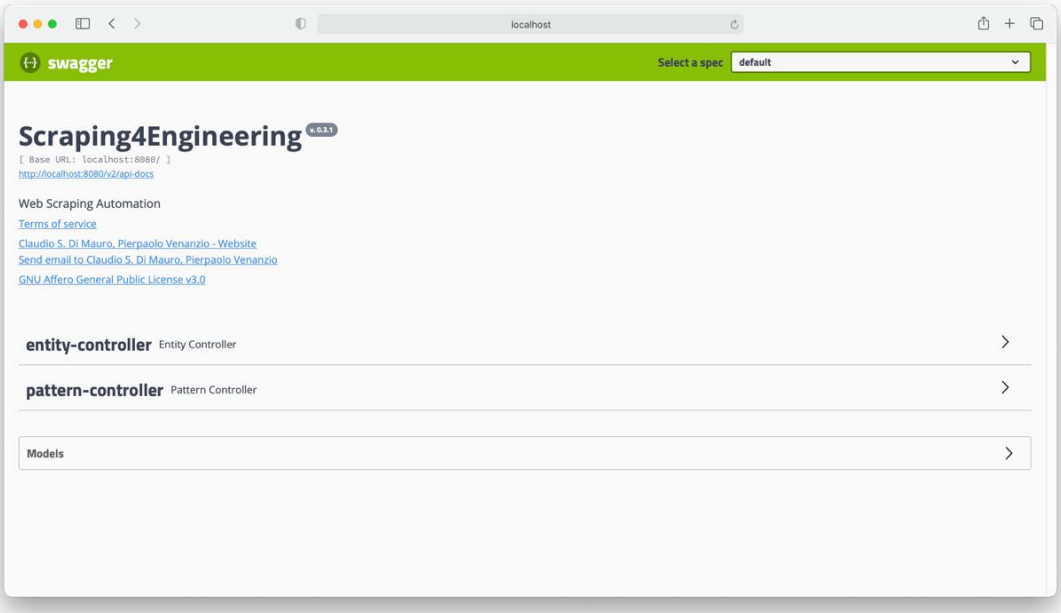

Fig. 18 - Swagger UI

Quella a video non è altro che l'interfaccia base di Swagger, all'interno della quale sono visibili i termini di servizio, le informazioni sugli sviluppatori, i contatti di questi ultimi e la licenza di rilascio. Sono altresì presenti delle sezioni tramite cui è possibile leggere la documentazione delle APIs e interagire con esse.

Nella scheda *models* sono presenti le informazioni riguardanti tutte quelle classi che sono definite come il "modello" della architettura MVC.

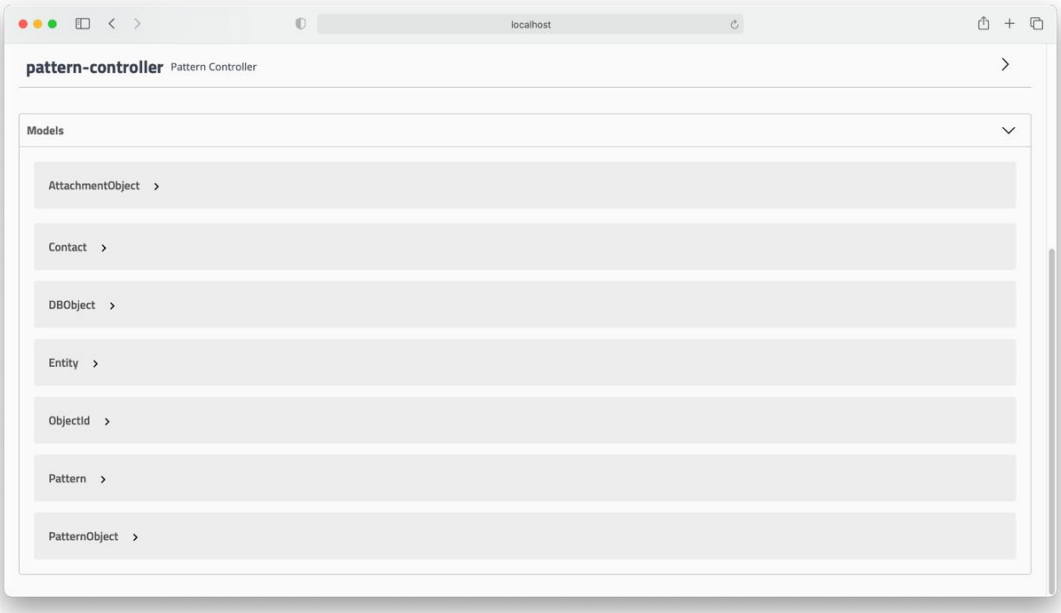

Fig. 19 – Models

Cliccando su una delle sotto-schede è possibile visionare la struttura di tali modelli e la loro documentazione, attributo per attributo, in modo da avere un quadro preciso di quello che è ciò che si va ad utilizzare.

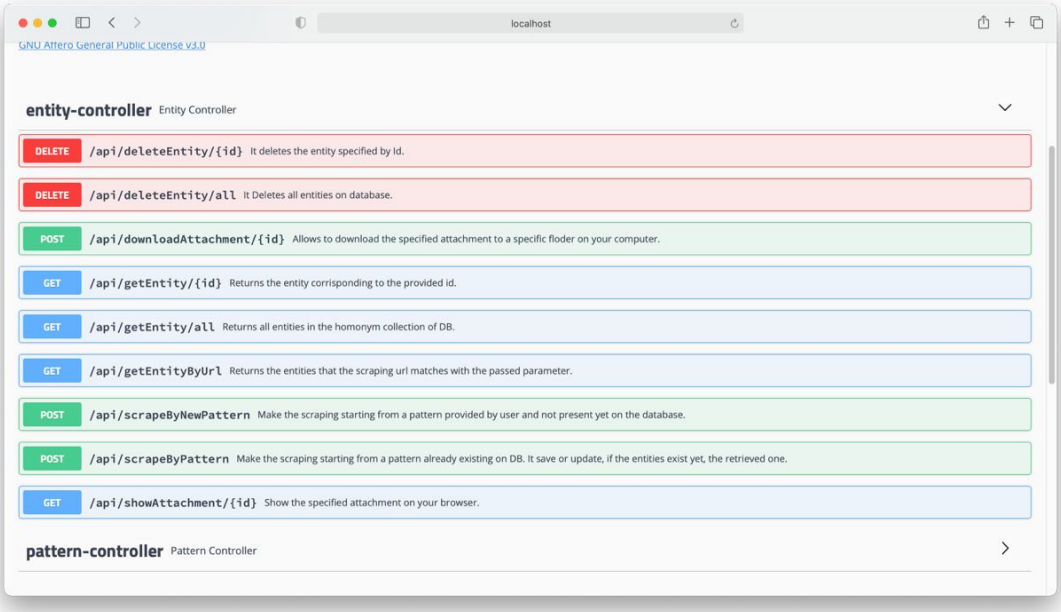

Fig. 20 - Entity-controller

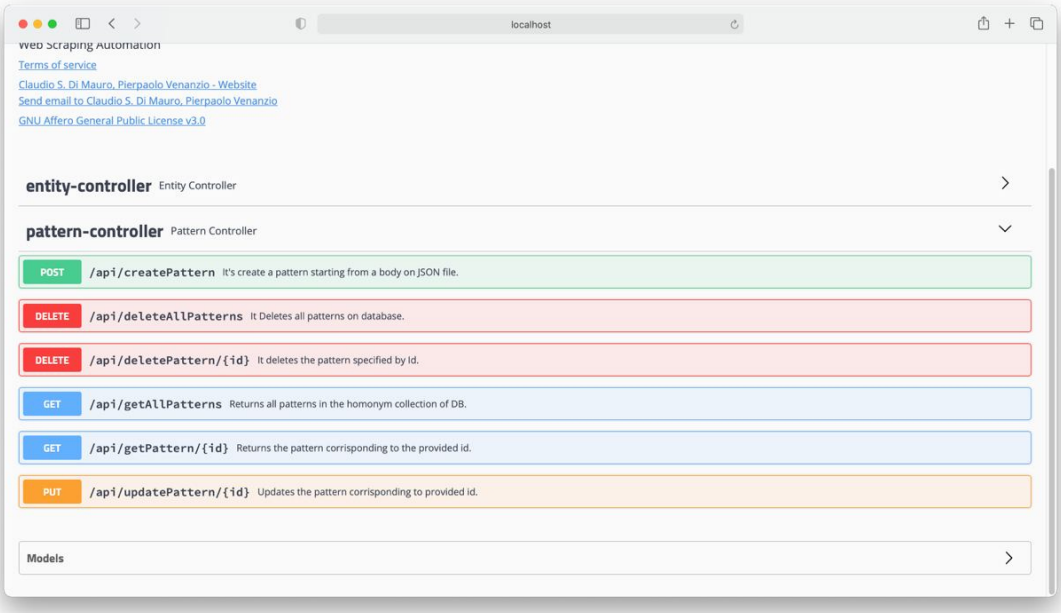

Fig. 21 - Pattern-controller

Entity-controller e **Pattern-controller** sono quelle schede che permettono di interfacciarsi con le APIs vere e proprie, andando a mostrare tutta una lista delle possibili chiamate che si possono effettuare.

Si supponga, a questo punto, di voler effettuare lo scraping delle informazioni presenti nella sezione “cronaca” del sito Ansa.it. La prima cosa da fare è creare il pattern che contenga le informazioni del DOM da estrapolare. Lo si fa chiamando api/createPattern e passando come corpo il pattern desiderato:

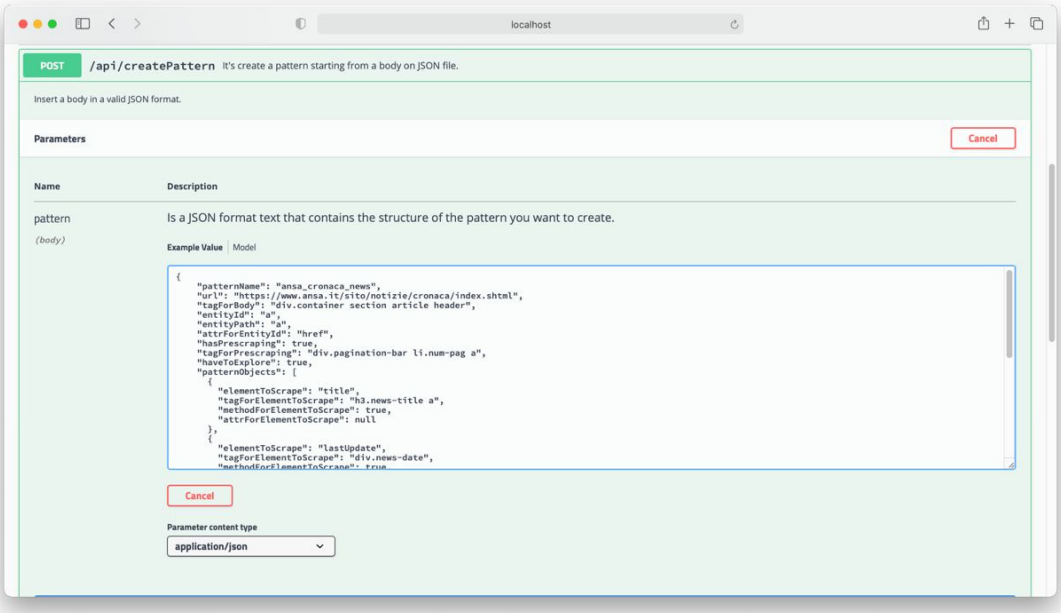

Fig. 22 - Creazione di un nuovo pattern

Eseguire la chiamata a questa API, produrrà un documento JSON sul database Mongo, contenente un pattern il cui id sarà “ansa_cronaca_news”. Per poter utilizzare questo pattern in uno scraping, ci si deve allora spostare nella scheda Entity-controller e chiamare la API api/scrapeByPattern, passando come body il nome del pattern che si vuole utilizzare, in questo caso proprio “ansa_cronaca_news”:

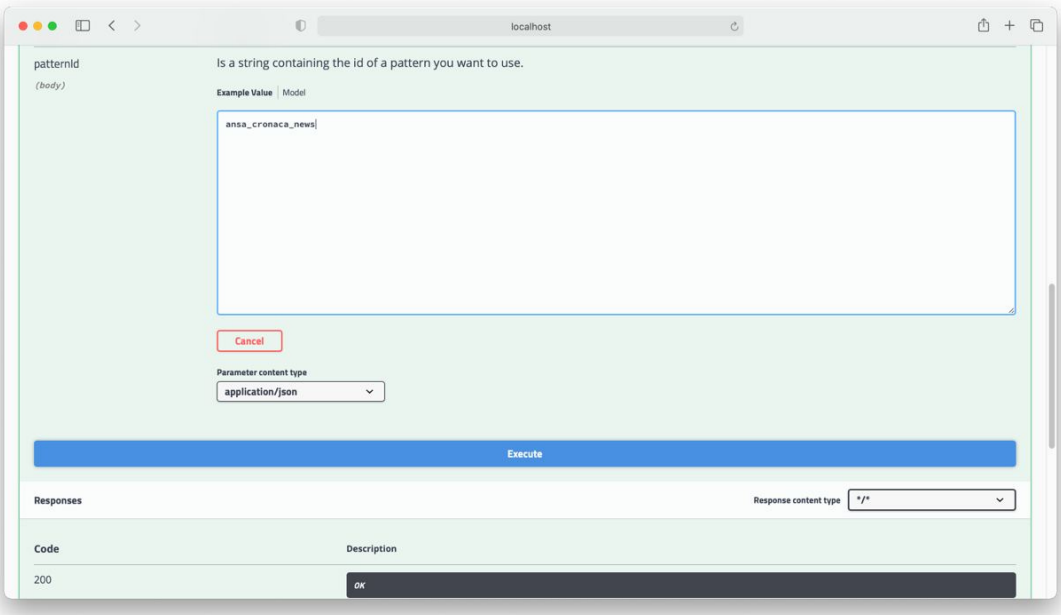

Fig. 23 - Esecuzione dello scraping

Cliccando sul pulsante “Execute” si avvierà lo scraping, il quale durerà un tempo variabile a seconda della quantità di informazioni da analizzare. Se il processo si concluderà in maniera corretta, verrà mostrato a video un *response body* riportante alcune informazioni salienti:

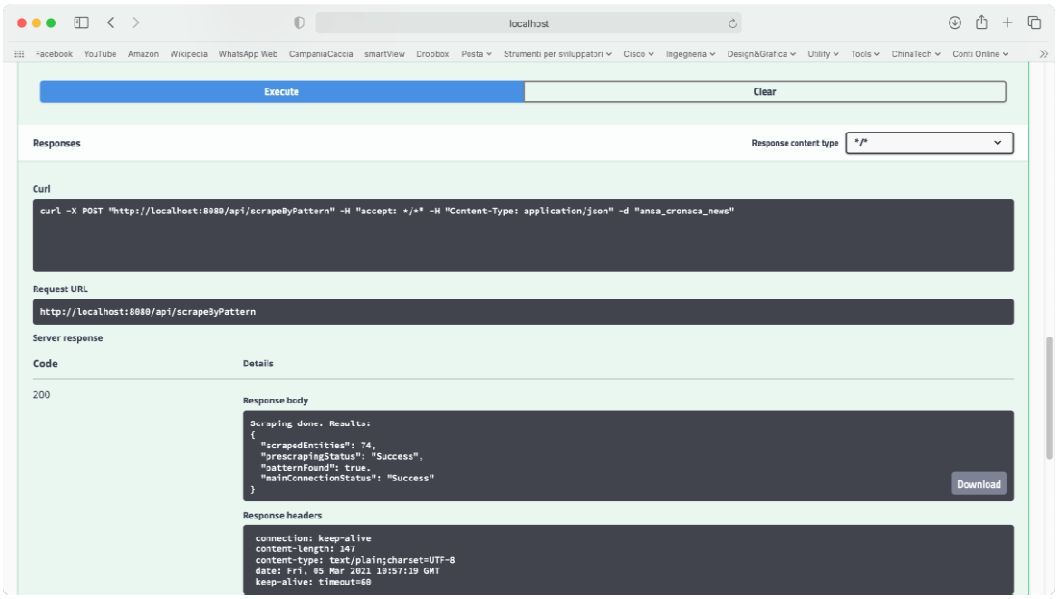

Fig. 24 - Response body

Anche nel caso in cui le operazioni si dovessero concludere in maniera anomala, verrà mostrato un response body e un codice di errore a seconda di quanto necessario.

L’operazione di scraping mostrata nelle figure precedenti ha come risultato quello di aver generato nel database un numero di documenti pari a quello delle entità recuperate:

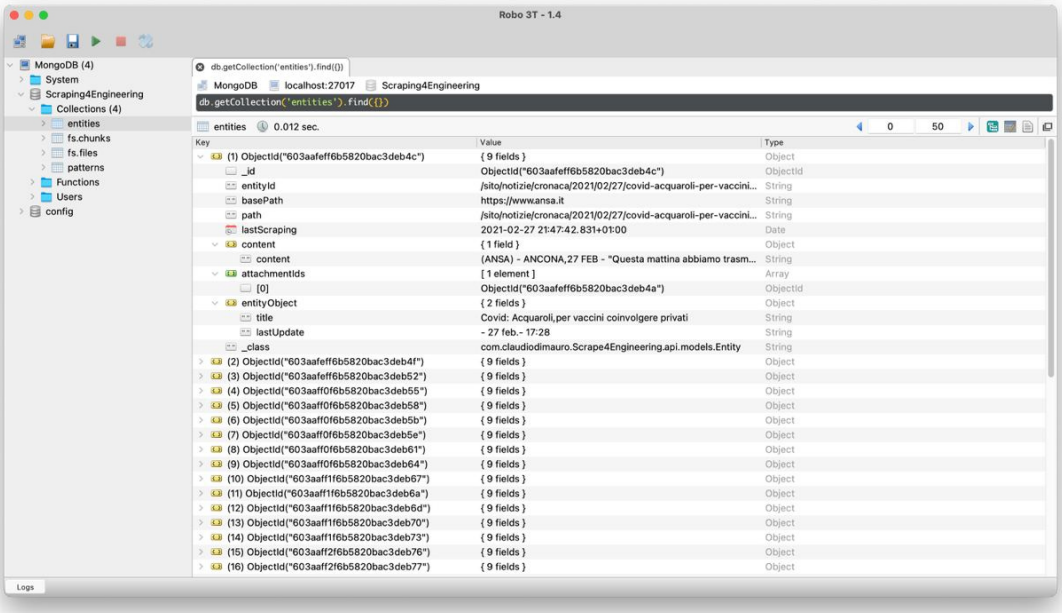

Fig. 25 - Le entity di Ansa.it

Le entità analizzate, nel caso di Ansa.it contengono degli allegati che a loro volta saranno storati nel database secondo lo schema adottato da GridFS e già spiegato nel paragrafo 2.2.4.

Se si trascura un po' tutta la parte sulle APIs di Pattern-controller, in particolare quelle relative al retrieving, alla cancellazione o all'upload dei patterns presenti sul database e ci si sofferma un po' di più su quelle di Entity-controller, si potrà vedere che lo scraping, oltre ad essere effettuato mediante un pattern creato in precedenza, può anche essere effettuato contestualmente alla creazione del pattern mediante la api/scrapeByNewPattern, una API che richiede come body il corpo di un pattern (eventualmente non ancora presente sulla base dati) e che, oltre a salvare il pattern passato, effettuerà l'analisi dei contenuti richiesti dal pattern stesso (qualora il pattern passato avesse un id uguale ad uno già presente sulla base dati, tramite un controllo, si ritornerà all'utente un messaggio di pattern non valido).

Questo modo di procedere è utile nel momento in cui si vuole effettuare per la prima volta l'analisi di una sorgente e non si ha già a disposizione un pattern: ciò consente di effettuare due operazioni in una sola volta. Il risultato prodotto sarà identico a quello mostrato nelle figure precedenti.

È di fondamentale importanza analizzare, e quindi testare, le APIs relative all'esposizione dei dati. In generale possono essere esposti sia i dati relativi ai patterns

sia quelli relativi alle entities, anche se fondamentalmente questi ultimi sono di maggiore rilevanza. L'esposizione dei dati sui patterns restituirà i pattern stessi, mentre l'esposizione dei dati relativi alle entità potrà esporre le entità trovate e/o gli allegati da esse ottenuti.

La gestione degli allegati è di particolare interesse. A seconda della API chiamata, si potrà scegliere se visualizzare gli allegati sul browser o addirittura scaricarli in una directory locale specificata dall'utente.

Le entità possono essere esposte tutte insieme o filtrate per id o per URL. Nell'esempio che segue, sarà chiamata la api/getEntity/all per mostrare tutte le entità trovate sul DB.

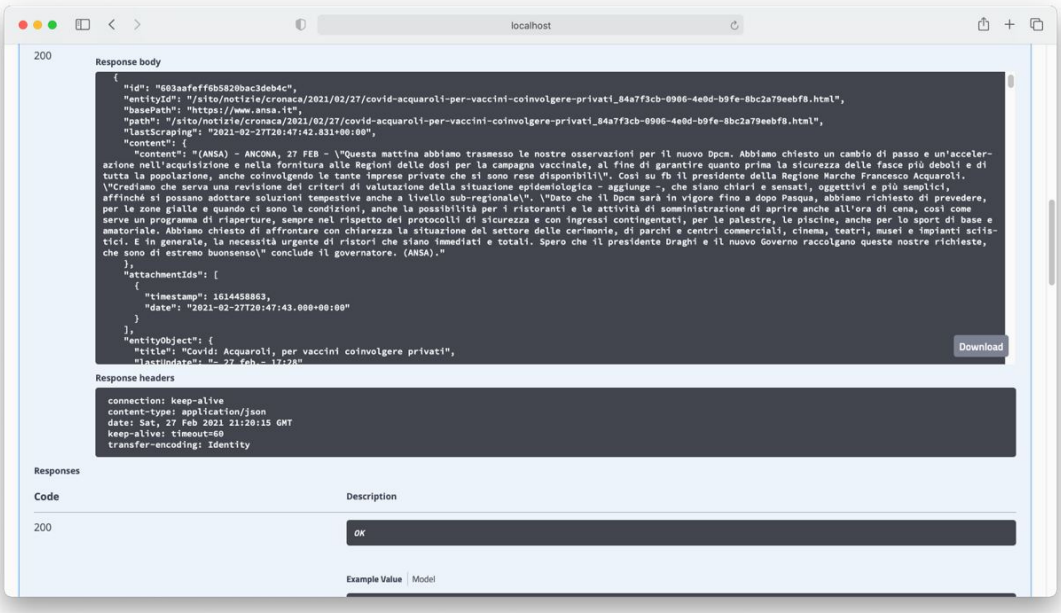

Fig. 26 - api/getEntity/all

Se l'operazione è conclusa correttamente, quello che sarà mostrato nella response è un array di JSON Objects contenente tutte le entità presenti nel database. Se avessimo scelto di utilizzare il filtro sugli id, sarebbe stato esposto un unico JSON Object con all'interno la entità corrispondente all'id specificato; se invece avessimo scelto di filtrare l'API sulle URL, sarebbero mostrate tutte quelle entità che hanno come URL di scraping quello specificato dall'utente.

Se un utente decide di scaricare uno degli allegati presenti sul database, può farlo chiamando api/downloadAttachment/{id}, dove al posto di {id} deve inserire l'identificativo presente su Mongo. Per poter recuperare l'id dell'entità, l'utente deve

ottenere i dati dell'entità specifica mediante una getEntity e da questi cercare l'id nel vettore **attachmentIds**, il quale contiene gli id di tutti gli allegati associati ad una entità.

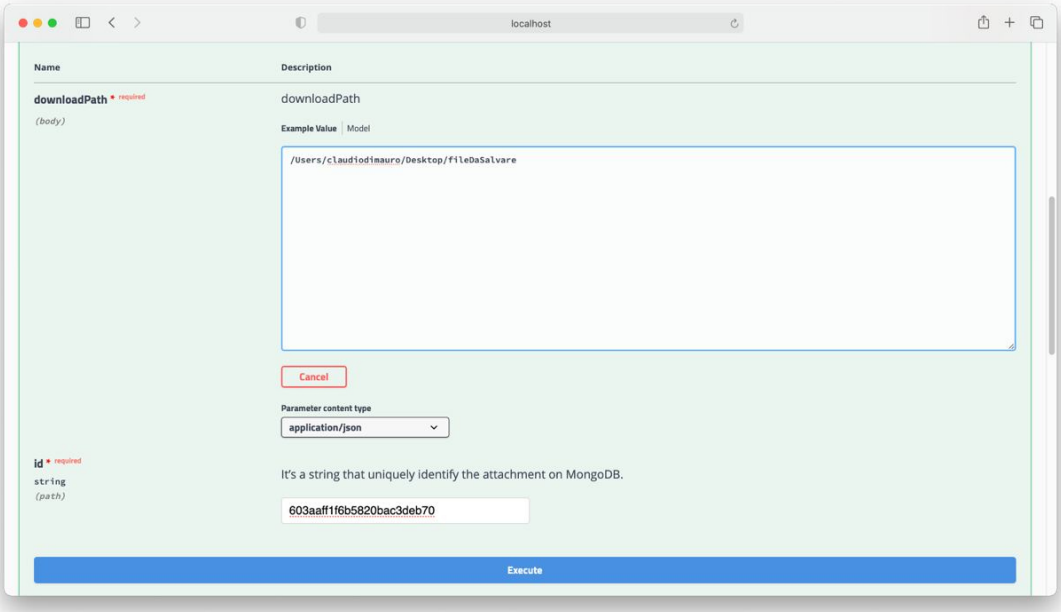

Fig. 27 - Download Attachments

downloadPath è un ulteriore parametro specificato dall'utente, per definire la directory locale in cui quest'ultimo intende salvare il file.

Se anziché salvare, l'utente vuole solo vedere il file senza scaricarlo, può farlo operando preferibilmente al di fuori di Swagger2. In particolare, può lanciare la chiamata alla API direttamente dal proprio browser, digitando nella barra degli indirizzi la stringa http://localhost:8080/api/showAttachment/{id}, dove al posto di {id} deve inserire l'identificativo presente su Mongo.

L'operazione, qualora l'id specificato trovasse una valida corrispondenza sul DB, mostrerà a video il file richiesto, sia esso un'immagine, un pdf, un video o anche un suono. L'esempio mostrato sulla figura seguente mostra un immagine ottenuta mediante le suddette azioni:

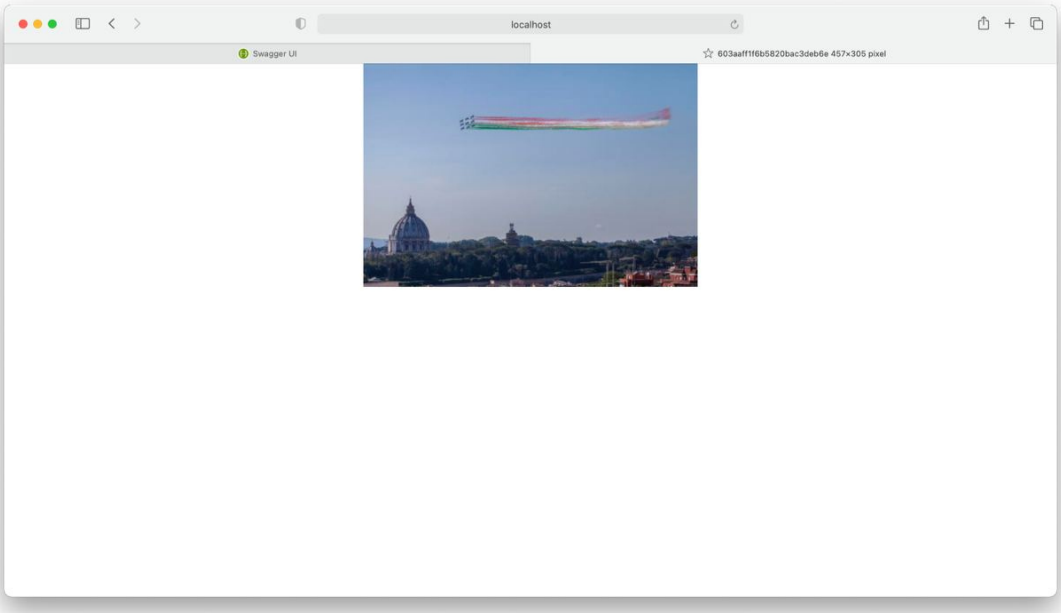

Fig. 28 - api/showAttachment/{id}

4.1.2 Dockerizzazione e deploy

Per questo progetto si è scelto di effettuare il deploy mediante "dockerizzazione". Questo sistema, come già specificato nel paragrafo in cui si è parlato di Docker, permette di effettuare una distribuzione del software molto più veloce e in maniera molto più efficiente.

Per tale motivo si è quindi provveduto a creare un'immagine Docker di tutto il pacchetto software e ad utilizzarla mediante un apposito container. Tale immagine è stata creata a partire dal pacchetto **jar** (Java ARchive), tramite cui è possibile raggruppare sotto un unico file eseguibile tutte le funzionalità del software sviluppato.

Per poter effettuare operazioni tramite Docker, è essenziale che sulla propria macchina sia installata una versione dello stesso, con relativo Docker Engine. A partire da codifica su linea di comando è possibile operare.

Il primo passo effettuato per creare una immagine del software progettato è stato quello di andare a scrivere un *Dockerfile*, all'interno del quale devono essere scritte tutte

quelle linee di codice che vanno a caratterizzare l'immagine. Nel caso in esame, il Dockerfile scritto è stato il seguente:

```
1.  FROM openjdk:12
2.  EXPOSE 8080
3.  ADD target/Scrape4Engineering-0.0.3-SNAPSHOT. jar app.jar
4.  ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Snippet 10 – Dockerfile

In questo file, quello che si è fatto è stato specificare con la keyword **`FROM`** la JDK utilizzata, la porta esposta (**`EXPOSE`**), il pacchetto di cui si vuole creare l'immagine (**`ADD`**) e il punto di partenza da cui si vuole che questo inizi a lavorare (**`ENTRYPOINT`**).

A partire da questo file è possibile lanciare da riga di comando la stringa tramite cui avviare la creazione dell'immagine. Nello specifico, bisogna posizionarsi nella cartella in cui è presente il Dockerfile e chiamare il comando *docker build*. Per il progetto presentato sarà:

```
docker build –t scrape4engineering:v0.0.3 .
```

Con **`–t`** si è specificato il tag tramite cui identificare l'immagine e la versione con cui questa deve essere distribuita.

A questo punto, una volta ottenuta l'immagine, si potrebbe già pensare di creare un container in cui farla girare, tuttavia è bene osservare che per poter lavorare correttamente, questa immagine ha bisogno di una istanza di MongoDB che possa interfacciarsi con il software contenuto al suo interno. In questo caso si può agire in due modi: il primo prevede che l'utente, qualora non ne fosse già provvisto, scarichi e faccia girare in un contenitore una immagine Docker di MongoDB e che la linki adeguatamente al container di *scrape4engineering* al fine di farle comunicare e lavorare; un secondo metodo, quello utilizzato per l'occasione, richiede invece l'utilizzo del file ***docker-compose.yml***, all'interno del quale scrivere tutte le dipendenze, i link e le caratteristiche che il contenitore generale deve inglobare per poter lavorare correttamente.

Ciò che si è fatto, allora, è stato scrivere il seguente docker-compose.yml:

```
1.  version: "3.1"
2.  services:
3.      mongoDB:
4.        image: mongo:latest
5.        container_name: mongoDB
6.        ports:
7.          - 27017:27017
8.      scrape4engineering:
9.        image: scrape4engineering:v0.0.3
10.       container_name: scrape4engineering
11.       ports:
12.         - 8080:8080
13.       links:
14.         - mongoDB
```

Snippet 11 – docker-compose.yml

Sfruttando il comando docker-compose up, chiamato da terminale nella cartella in cui il docker-compose.yml è contenuto, si avvia la creazione di un unico container, al cui interno sono inglobati sia il container per l'immagine *scrape4engineering*, sia quello per *mongo*. MongoDB gira su un container che espone la porta 27017, a seguito del mapping fatto alla riga 7; scrape4engineering espone la porta 8080. Con la keyword **`links`** si è effettuato il collegamento tra i due container in modo da permetterne il corretto interfacciamento.

Quello che succede in questo caso è che l'utente che non dispone di un container pre-installato per Mongo, non ha bisogno di creare due container separati, ma semplicemente con un unico comando può ottenere i due container che servono, già linkati e pronti a comunicare tra loro. Tramite il docker-compose vengono automaticamente scaricate le immagini di Scrape4Engineering e di MongoDB dal **Docker Hub**[14].

Una volta avviato il container, il software sarà fruibile collegandosi dal proprio browser al link *http://localhost:8080/swagger-ui.html*.

Una primissima fase di deploy ha previsto l'utilizzo della piattaforma Docker Hub per favorire la distribuzione dell'immagine anche ai membri interni dell'Azienda che ha ospitato il tirocinio. Inizialmente il sorgente è stato posto su un repository privato di GitHub; tale repository è stato poi reso pubblico a fine sviluppo, dato che il software viene rilasciato sotto licenza open-source: è parso allora ovvio rendere disponibile il codice a tutta la community.

[14] **https://hub.docker.com** è la piattaforma ufficiale tramite cui è possibile condividere con la community le immagini dei propri software dockerizzati.

4.1.3 Possibili miglioramenti

Per i tempi e le modalità di sviluppo, il progetto deve necessariamente considerarsi migliorabile. La versione rilasciata ai fini del completamento del tirocinio è solo una versione beta che, per quanto funzionante, va perfezionata in diversi suoi punti.

Una primissima operazione da fare, sarebbe quella di portare avanti una importante fase di debugging, eventualmente fatta da sviluppatori esperti, che permetta di scovare anomalie da correggere nel funzionamento. Sicuramente quella effettuata in fase di testing ha permesso di portare alla luce alcuni malfunzionamenti che si è provveduto a sistemare, tuttavia è comunque poca cosa, rispetto a quella che potrebbe fare un qualcuno specializzato in questo.

Una generale pulizia del codice e una ottimizzazione delle nomenclature sarebbe opportuna al fine di rendere fruibile il codice anche ad altri sviluppatori. Nonostante si sia cercato di essere quanto più coerenti possibili nell'assegnare nomi a variabili e metodi, proprio per favorirne la comprensione, ci si è resi conto che in determinati punti andrebbero apportate delle modifiche sugli stessi. È chiaro che ciò non comporterebbe né una modifica strutturale, né una modifica del comportamento finale del software.

In stretta correlazione con ciò, un ulteriore miglioramento si dovrebbe avere circa la documentazione, sia lato sviluppo che lato utente: lato sviluppo una buona **Javadoc**[15] consentirebbe ad eventuali sviluppatori successivi di comprendere le funzioni dei metodi e la struttura del codice; lato utente, un miglioramento nella documentazione delle APIs permetterebbe un uso più saggio dei pattern di cui si è ampiamente parlato, assicurando quindi una maggiore comprensione dei parametri HTML da passare affinché lo scraping funzioni.

Un ulteriore miglioramento che potrebbe essere aggiunto al progetto potrebbe essere quello di trattare anche i contenuti codificati come **Base64**, un particolare tipo di codifica che prevede di tradurre le stringhe di dati binari in caratteri ASCII[16]. È raro, ma non impossibile, imbattersi in pagine Web che utilizzano tale codifica per linkare gli al-

[15] **Javadoc** è un applicativo software sviluppato dalla SUN Microsystems, e inglobato nel *Java Development Kit* (comunemente abbreviato n *JDK*), che permette la generazione automatica della documentazione del codice sorgente scritto in linguaggio Java, a partire da alcune annotazioni tramite le quali evidenziare i punti da documentare. Il software si occupa in maniera automatica di generare una pagina HTML riportante la documentazione nel classico stile di quella ufficiale Oracle.

[16] **ASCII**, *American Standard Code for Information Interchange*, è uno standard datato, ma comunque ancora molto in uso, per la codifica dei caratteri a 7 bit. Sta via via venendo sostituito da **UTF-8**.

legati presenti in essi; con il software oggetto di questo elaborato, non è possibile prelevare tale tipo di allegati e, pertanto, sarebbe opportuno implementare quelle funzionalità che permettano di farlo. Volutamente si è scelto, almeno in una fase iniziale, di non tener conto di allegati presentati in tale codifica.

La gestione di un versioning sugli allegati ottenuti dallo scraping e salvati sul database potrebbe aiutare a tenere traccia di eventuali aggiornamenti apportati a questi ultimi. Attualmente il software, nel momento in cui lo scraping preveda degli allegati, quello che fa è andare ad eliminare quelli già presenti sulla base dati e ricaricare quelli trovati. Nonostante si fosse consapevoli del fatto che questa operazione di cancellazione e riscrittura comporti un dispendio computazionale leggermente maggiore, si è deciso di operare in questo modo per uno scopo ben preciso. Lo scraping sugli allegati, se non correttamente gestito, porta ad una saturazione della base dati, poiché ad ogni analisi effettuata su una pagina gli allegati verrebbero presi e aggiunti come duplicati sul DB, per cui quello che si è fatto è stato controllare il tutto affinché tali duplicati non fossero generati. Ci si è però accorti del fatto che, qualora un allegato di una pagina mantenesse lo stesso nome, ma aggiornasse il suo contenuto interno (ad esempio l'aggiornamento di un pdf), sulla base dati verrebbe mantenuta comunque una versione non aggiornata. È chiaro, quindi, il perché si è scelto di cancellare e poi ricaricare gli allegati analizzati.

Ciò che potrebbe ulteriormente essere migliorato sono anche le response che si ottengono nel momento in cui una delle operazioni viene effettuata. Le response rappresentano quelle informazioni che permettono sia all'utente che allo sviluppatore di capire, anche in maniera approssimativa, dove può essersi generato un errore e provvedere quindi a correggerlo. In associazione ai file di log, le response rappresentano uno strumento molto valido per evitare comportamenti errati. Attualmente si sta già lavorando al miglioramento di tale feature.

4.2 Conclusioni

4.2.1 Sviluppo e usi futuri

Il progetto sviluppato per il lavoro di tirocinio in oggetto è un qualcosa che si presta ad avere buone potenzialità, soprattutto perché arriva ad abbracciare quello che è lo scraping dinamico, un qualcosa fin ora realizzato solo da grandi sviluppatori. Questo

software, però, fa un qualcosa che gli altri non fanno, ossia garantire all'utente la possibilità di scegliere nel dettaglio ciò che si vuole analizzare e ciò che si vuole trascurare.

In futuro per questo software si potranno fare tanti accorgimenti, uno dei quali è sicuramente una buona interfaccia grafica che possa permettere all'utente di operare in maniera più diretta e magari anche più automatizzata nella definizione di quelli che saranno i pattern da utilizzare.

L'uso che se ne potrà fare sarà principalmente aziendale, dato che lo scraping può essere comodo per la ricerca di informazioni contro competitors o per fini statistici; ciò non esclude, però, anche un utilizzo da parte di utenti privati o addirittura un utilizzo accademico, ad esempio per fini di ricerca: riuscire a collezionare tutta una serie di dati da fonti eterogenee può sicuramente coadiuvare lo studio e lo sviluppo.

4.2.2 Considerazioni finali

È opportuno, in fase conclusiva del presente elaborato di tesi, andare a fare qualche breve considerazione.

Come spesso accade, il tirocinio curricolare nasconde una doppia finalità: portare un lavoro valido all'azienda che lo ospita e fare in modo che lo studente che lo mette in pratica abbia un primo approccio con un qualcosa di reale.

Nel caso del progetto in esame, è stato esattamente così. L'idea di partenza era lo sviluppo di questo software che riempisse i cataloghi MedITech, quindi un qualcosa che potesse effettivamente essere usato dalla Engineering e, grazie a tale sviluppo, dare allo studente la possibilità di imparare qualcosa di nuovo.

Questo progetto ha consentito a chi lo ha sviluppato di imparare nuovi framework e nuove librerie (vedi Spring Boot e JSoup), di interfacciarsi con una variante Java (la Enterprise Edition) che a livello accademico non viene toccata (solitamente si utilizza la Standard Edition), di conoscere un nuovo tipo di basi dati, quelli non relazionali, e di capirne il funzionamento sfruttandone uno (MongoDB) per il salvataggio dei dati; ha avviato lo studente alla conoscenza dei nuovi metodi e delle nuove infrastrutture di lavoro (si pensi all'architettura a microservizi e a Docker).

Il tutto si è svolto in maniera abbastanza autonoma, sotto la guida di un tutor aziendale che si è impegnato a fornire qualche riferimento e qualche dritta per

l’implementazione delle funzionalità, ma che comunque ha lasciato ai tirocinanti la libertà delle scelte progettuali e delle tecnologie abilitanti.

Il self-learning perpetrato per imparare cose mai viste durante il corso di studi ha permesso anche di acquisire una parte di quelle soft skills tanto richieste sul posto di lavoro: riuscire a portare a termine un qualcosa, sfruttando tecnologie mai viste prima, ha dato tanta soddisfazione e tanta gratificazione. Si è imparato facendo.

RINGRAZIAMENTI

3102 giorni. 8 anni, 5 mesi e 29 giorni[17]. Tanto è il tempo che è passato mentre aspettavo questo momento. Ora il giorno è arrivato e non mi sembra vero.
Chi mi conosce, sa benissimo le difficoltà incontrate durante questo percorso tortuoso e tutte le volte che ho pensato di mollare.
Col tempo ho visto svanire le mie speranze e quelle di chi, quando ho cominciato, credeva in me. Sembrava una via senza uscite, ma grazie all'aiuto e allo sprono di tanti ho perseverato… e ce l'ho fatta!

Mi sembra allora doveroso dedicare questo spazio del mio elaborato alle persone che hanno contribuito con il loro supporto, amorevole e soprattutto paziente, a farmi arrivare fino a questo punto.

In primis, un ringraziamento speciale al mio relatore, il prof. Ritrovato Pierluigi, per la sua immensa bontà, nonostante le mie continue richieste di informazioni.

Il ringraziamento più grande va ai miei genitori: senza di loro oggi questa tesi non esisterebbe. Hanno sempre appoggiato le mie scelte, mi hanno sempre consigliato al meglio, hanno fatto tanti sacrifici per farmi studiare e non me lo hanno mai fatto pesare, nonostante gli anni di fuoricorso. Spero che un giorno potrò ricambiare i loro sacrifici e, anche se in ritardo, di avergli dato una piccola soddisfazione, oggi.

Ringrazio mia nonna Rosa, la mia seconda mamma, colei che mi ha cresciuto e mi ha sempre indicato la giusta via. Il suo amore mi ha dato la forza per affrontare la vita intera, non solo quella universitaria.

Ringrazio mia sorella che nonostante i continui litigi ha sempre voluto il meglio per me, sopportando tutte le volte che mi sono innervosito quando si toccava il tasto dolente "università". È lei che mi ha fatto uno dei regali più belli della vita: il mio nipotino, Felice, che tante volte ha riempito le mie giornate, anche e soprattutto durante lo studio.

Ringrazio Rosanna, fidanzata, amica, confidente: non potevo chiedere di meglio. Con la sua pazienza infinita mi ha sopportato durante gli ultimi anni di università, i più

[17] 3102 giorni sono quelli trascorsi dal primo giorno di università, il 24 settembre 2012, al giorno della laurea, il 23 marzo 2021.

difficili, senza mai abbandonarmi, senza mai farmi sentire solo, consigliandomi sempre il meglio. Ha saputo rassicurarmi quando tutto era nero. E continua a farlo ancora oggi.

Ringrazio i miei zii, Anna, Peppe, Mario e Antonio. Non sono solo i miei zii, ma sono parte attiva della mia vita, sono i miei amici, i miei confidenti, sono coloro che hanno sofferto e gioito insieme a me per le bocciature e gli esami superati. Su di loro so di poter contare sempre, qualsiasi cosa succeda.

Ringrazio Lina e Maurizio, i miei suoceri, che mi hanno accolto in casa come un figlio e mi hanno sempre rassicurato nei momenti di sconforto. Le loro parole dolci mi saranno sempre d'aiuto.

Voglio ringraziare anche i miei amici, quelli veri, che mi hanno accompagnato, che mi hanno supportato e che hanno significato qualcosa per me in questi anni di università.

Ringrazio Francesco (Alfieri), per essere un amico sincero, per essere quell'amico sempre presente, anche se non ci vediamo spesso.

Ringrazio Pasquale (Piscopo), perché i momenti passati insieme sono stati tra i più belli e divertenti della mia vita. Le sue telefonate pomeridiane hanno spezzato quotidianamente la monotonia dello studio e ha saputo farmi scaricare lo stress che questo mi portava.

Un grazie anche a chi è stato meno attivo nella mia vita universitaria, ma non per questo meno presente: Pasquale (Sannino), mio eterno compagno di banco e collega di tante avventure e tante partite del Napoli; Francesco (Gallo), uno che c'è, anche se si vede poco.

Ringrazio Andrea, Raffaele e Riccardo, gli amici che hanno riempito le mie giornate quando ero alla Federico II, quelli che hanno condiviso con me parte del percorso e con cui ho passato momenti bellissimi e bruttissimi (vedi Teoria dei Segnali e Metodi Matematici)… un giorno, però, all'improvviso… "l'avetta fa 'na paliat"!

Il mio percorso universitario ha visto tanti rallentamenti. Una delle pause più belle, però, è stata quella relativa alla Apple Developer Academy. È d'obbligo, allora, spendere due parole di ringraziamento per i due amici che hanno fatto sì che quei nove mesi fossero bellissimi: Felice e Fulvio. Grazie, perché l'esperienza con voi è stata ricca di gioie e divertimenti, grazie perché mi avete sostenuto quando studiavo per un esame, contestualmente all'Academy.

Infine, ma solo per una questione “cronologica”, voglio ringraziare Stefano. Ho conosciuto i suoi appunti prima di lui: alla Facoltà di Ingegneria di UniSA la fama dei suoi quaderni lo precede.

Stefano si è dimostrato un amico vero, mi ha aiutato quando mi sono trasferito a Salerno, dove non conoscevo nessuno e tanti corsi mi erano ignoti. È stato l’unico che veramente lo ha fatto col cuore e in maniera spassionata, senza mai scocciarsi. A lui devo tanto e ringraziarlo nel mio elaborato era la minima cosa che potessi fare.

A chi c’è stato, a chi non c’è stato, a chi ha condiviso qualcosa con me, a chi mi ha voluto bene, a chi me ne vuole ancora, a chi ha creduto in me fino alla fine, a chi ci ha creduto per un po’ ma poi ha smesso di farlo, a chi mi ha spinto a non mollare, a chi mi ha consigliato di trasferirmi dalla Federico II all’Università di Salerno. A tutti voi, GRAZIE.

#callmeengineer

Bibliografia e sitografia

[1] *Pagina Wikipedia:* https://it.wikipedia.org/wiki/Web_scraping

[2] *Pagina Web:* https://www.ionos.it/digitalguide/siti-web/programmazione-del-sito-web/che-cose-il-web-scraping/

[3] *Pagina Web:* https://www.ionos.it/digitalguide/siti-web/programmazione-del-sito-web/document-object-model-dom/

[4] *Pagina Wikipedia:* https://it.wikipedia.org/wiki/Crawler

[5] *Pagina ufficiale:* https://microservices.io/

[6] *Pagina Amazon AWS:* https://aws.amazon.com/it/microservices/

[7] *Pagina ufficiale:* https://jsoup.org/

[8] *Pagina Web:* https://code.tutsplus.com/it/tutorials/a-beginners-guide-to-http-and-rest--net-16340

[9] *Pagina Web:* https://dzone.com/articles/introduction-to-mongodb-with-java

[10] *Pagina Web:* https://it.wikipedia.org/wiki/MongoDB

[11] *Pagina ufficiale:* https://swagger.io/

[12] M. Ahmadi, H. J. Bhatti, B. B. Rad, *An Introduction to Docker and Analysis of its Performance*, in "IJCSNS International Journal of Computer Science and Network Security", VOL. 17 No. 3, 2017, pp. 228-223

[13] *Pagina Web:* https://it.wikipedia.org/wiki/Git_(software)

[14] *Documentazione ufficiale:* https://git-scm.com/about

[15] *Pagina ufficiale:* https://github.com/

[16] *Documentazione ufficiale:* https://mongodb.github.io/mongo-java-driver/4.1/apidocs/mongodb-driver-sync/index.html

[17] P Chiuri, M. Kalelkar, D. Kalelkar, *Implementation of Model-View-Controller Architecture – Pattern for Business Intelligence Architecture,* in "International Journal of Computer Applications", VOL. 102 No. 12, 2014, pp. 16-19