UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche e Matematiche

CORSO DI LAUREA IN INFORMATICA

Applicazione Web per Dati Neuroscientifici: Generazione Dinamica di un Front End di Gestione

Relatore

Prof. Riccardo Martoglia

Laureando

Matteo Vanzini

ANNO ACCADEMICO 2018/2019

Sommario

Introduzione

Nel corso di questo elaborato si parlerà della progettazione di un database e della relativa interfaccia web per la gestione di dati in ambito neuroscientifico. La realizzazione e lo sviluppo del suddetto progetto sono stati concepiti e calibrati sulla base delle richieste di ricercatori, professori e dottori i quali, avendo aderito ad una ricerca che si propone di condurre uno studio circa il Disturbo Cognitivo Lieve, o Mild Cognitive Impairment (MCI), necessitavano di un database che consentisse loro l'archiviazione dei dati raccolti e di un'interfaccia per la gestione degli stessi.

Più precisamente, la struttura della trattazione prevederà: un primo capitolo riguardante le specifiche richieste dall'equipe di cui sopra e contenente inoltre un approfondimento teorico circa i componenti della web application; un secondo capitolo contenente lo studio delle tecnologie sul mercato, l'approfondimento di tutte le possibili soluzioni inizialmente reputate eventualmente adatte alle richieste della stessa equipe, ed i motivi che hanno portato alle scelte progettuali poi intraprese; un terzo capitolo riguardante la progettazione dell'applicazione web; infine, un ultimo capitolo che tratterà l'implementazione della stessa. Inoltre, si avrà cura di presentare sempre le motivazioni circa le scelte progettuali intraprese.

Si tenga infine presente il fatto che la suddetta Web Application prende il nome di MCIBase. Dunque, nel corso della trattazione, ci si riferirà ad essa con questo nome.

Parte I

Studio delle tecnologie utili alla realizzazione della Web Application

Capitolo 1

Concetti teorici

1.1 Il caso di studio

Il progetto di cui tratta questo elaborato nasce da uno studio sull' MCI a cui professori, dottori e ricercatori stanno attualmente partecipando.

Gli obiettivi del suddetto studio sono molteplici. In primis si occupa di studiare le alterazioni biologiche e funzionali in pazienti con MCI, con l'ulteriore scopo, inoltre, di individuare possibili marcatori prognostici di manifestazione clinica e progressione a demenza. Infine, si propone di valutare l'effetto di possibili fattori di rischio ambientali e nutrizionali nell'insorgenza, manifestazione clinica e progressione a demenza.

Per raggiungere i suddetti scopi, i partecipanti allo studio, hanno richiesto, in primis, la creazione di un database centrale a cui tutti i partecipanti potessero accedere da diverse locazioni, ed infine, un'interfaccia con cui gestire i dati sul database.

In questo capitolo andremo a trattare gli aspetti teorici su cui la web application MCIBase, nonché la soluzione che si è pensato di realizzare a seguito delle richiese dell'equipe, basa le sue fondamenta. In particolare, andremo ad analizzare nella sezione 1.2 le richieste su cui il progetto si è sviluppato e le specifiche tecniche su cui MCIBase è stato creato. Infine, nella sezione 1.3 andremo a trattare la teoria che sta alla base delle applicazioni web e i concetti fondamentali utili allo sviluppo di una di queste.

1.2 Specifiche tecniche

Per poter presentare il lavoro fatto durante tutto lo svolgimento del progetto, è necessario presentare le specifiche forniteci per lo sviluppo di MCIBase.

In particolare, le suddette specifiche sono state strutturate a partire dalle richieste del gruppo medico professionale formato da (in ordine alfabetico): Dott.ssa Chiara Carbone, RUTD Tommaso Filippini, prof. Giuseppe Pagnoni, prof. Marco Vinceti, e Dott.ssa Giovanna Zamboni.

Dopo diversi incontri con questo gruppo di lavoro, sulla base delle esigenze di ogni componente dell'equipe, sono state identificate le necessità sulla base delle quali sono state costruite le specifiche del progetto:

- Necessità di poter archiviare i dati raccolti nei diversi esami
- Necessità di poter creare strutture ripetutamente annidate
- Necessità di poter utilizzare strutture diverse per diversi esami e di poterle eventualmente successivamente modificare
- Necessità di poter, per determinati esami, caricare interi file CSV da archiviare direttamente

- Necessità di poter archiviare e successivamente visualizzare i dati collezionati tramite un'interfaccia funzionale
- Necessità di poter modificare, aggiungere od eliminare i dati raccolti ed inseriti nel Database
- Necessità di poter scaricare, anche parzialmente, i dati in un file CSV
- Necessità di poter distinguere diversi livelli di utente e, di conseguenza, diversificare le azioni associate consentite
- Necessità di poter inserire documenti incompleti, contando sul fatto che specifici campi sarebbero stati riempiti automaticamente sulla base di determinati vincoli forniti

Di conseguenza sono state costruite specifiche che soddisfacessero questo bisogni. Fin dal principio, a causa della richiesta di un sistema di archiviazione e di un'interfaccia tramite il quale effettuare determinate operazioni sui dati archiviati, il progetto si è diretto verso una web application [1]. Quest'ultima, come vedremo nella prossima sezione, dà modo di rispettare tutte le richieste avanzate dall'equipe. Come detto nell'introduzione della trattazione l'applicazione web in questione ha preso il nome di MCIBase.

1.3 Concetto di Web Application

Come detto nella sezione 1.1, in questa parte dell'elaborato andremo ad analizzare gli aspetti teorici di una web application e le basi fondamentali su cui viene generalmente sviluppata.

Quando si parla di applicazione web si intende generalmente di un'applicazione accessibile mediante una connessione internet, più nello specifico, si tratta di un servizio che un server offre ad un client che si connette ad esso. Questo particolare sistema si basa sull'architettura client-server. Comunemente, il servizio offerto non è altro che un'interfaccia che agisce direttamente su determinati dati che risiedono in un server. In altre parole, una web application è uno strumento che permette ad un client,

in comunicazione con un server, di eseguire operazioni tramite un'interfaccia utente sui dati archiviati sul server stesso.

Un'applicazione web può presentarsi con diverse strutture ed organizzazioni logiche. Dal punto di vista dell'informatica teorica è possibile riconoscere una strutturazione dell'architettura di sviluppo su più livelli, mappando a livello fisico ed infrastrutturale un sistema informatico. Tale struttura è chiamata architettura multi-tier.

Nella prossima sezione andremo ad analizzare un particolare architettura, riconducibile al gruppo delle architetture multi-tier, l'architettura three-tier [2].

1.3.1 Architettura three-tier

Da un punto di vista ingegneristico, l'architettura multistrato è una struttura software in cui le diverse funzionalità sono logicamente separate e divise in livelli o strati comunicanti tra loro. Nello specifico, ogni livello è in comunicazione con lo strato precedente e con quello successivo, in modo da offrire un servizio ai livelli circostanti.

L'architettura di cui tratteremo nello specifico è quella rappresentata in Figura 1.3.1. Come è possibile notare, è composta da tre strati (architettura three-tier), i quali prendono il nome di interfaccia utente o User Interface (UI), logica funzionale e database[4].

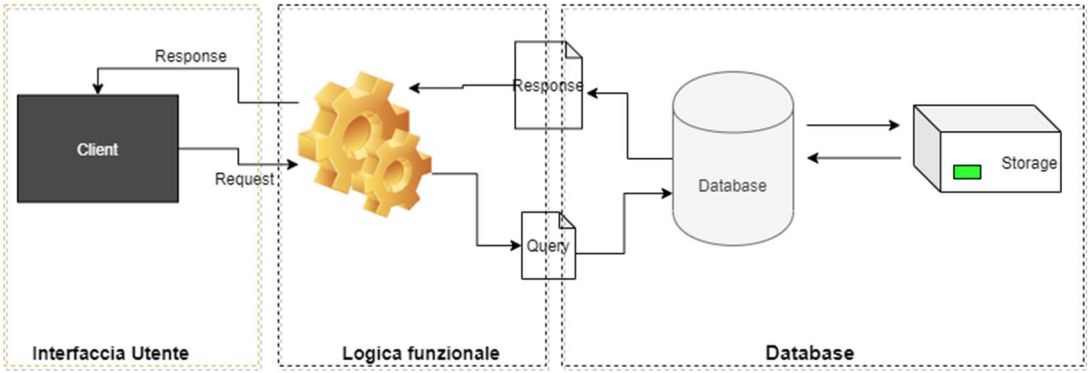

Figura 1.3.1: architettura three-tier.

Si proseguirà ora con l'analisi dei livelli:

- Interfaccia utente: questo strato è genericamente un software che permette ad un utente di interagire con una macchina. Nel caso del progetto che stiamo trattando, permetterà all'utente di caricare i dettagli degli esami e di salvarli nel livello più basso - il database - di cui tratteremo successivamente.
- Logica funzionale (o di Business): questo componente si riferisce al nucleo dell'applicazione. La logica, infatti, è il livello che gestisce l'interfaccia utente, e in base alle operazioni effettuate, agisce sul database. Se generalmente all'interfaccia viene attribuito il nome di front end, in quanto lato che si mostra all'utente, la logica funzionale rappresenta il back end, ovvero il componente che mette in comunicazione la UI e il database. [3]
- Database: una base di dati o database è la parte più importante di un qualsiasi sistema. Un database, infatti, è uno strumento grazie al quale è possibile memorizzare ed estrarre informazioni. In questo particolare caso con termine "database" si intende una combinazione di due livelli, che vengono solitamente mantenuti separati. I livelli di cui parleremo sono quello logico e quello fisico. Quest'ultimo si riferisce alla parte hardware, ovvero i supporti di memorizzazione che contengono i dati stessi. L'altro livello invece, quello logico, è più comunemente chiamato database management system (DBMS): un'applicazione che consente la manipolazione dei dati in modo efficiente. Un DBMS si occupa, quindi, di inserire, eliminare, modificare ed estrarre dati dal livello fisico.

Capitolo 2

Studio delle tecnologie

2.1 Introduzione allo studio tecnologico

Nel capitolo precedente sono state trattate le richieste dell'equipe, nonché le specifiche su cui il progetto è stato sviluppato. Successivamente sono stati analizzati i concetti teorici su cui un applicazione web è strutturata. In questo capitolo, invece, andremo ad analizzare, relativamente ai componenti di cui abbiamo trattato nel precedente, le

tecnologie sul mercato, presentando a grandi linee le differenze tecniche e trattando nello specifico i servizi più importanti tra quelli riportati. La trattazione inizierà con lo studio dei database, per poi proseguire con i framework ed infine con il linguaggio di programmazione.

2.2 Database

Come accennato, il primo passo è consistito nell'identificare le possibili tecnologie di database idonee a supportare questo progetto. Verranno qui di seguito elencate diverse strutture di database, ciascuna con le proprie caratteristiche di archiviazione, organizzazione e manipolazione dei dati, con l'obiettivo di identificare quella maggiormente adatta alle suddette esigenze.

Navigational model:

- La modalità primaria di ricerca di un record è quella mediante relazioni con altri record
- In una ricerca complessa, come la ricerca di una lista, i record vengono cercati sequenzialmente, uno alla volta
- Usabilità critica a livello di codice
- Il codice dipende dalla struttura dei dati: se uno cambia deve farlo anche l'altro

Relational model:

- I database basati su questo modello usano principalmente il linguaggio SQL per archiviazioni e interrogazioni
- Principalmente rappresentato mediante tabelle, composte da tuple associate, a volte, ad altre tabelle mediante chiavi esterne
- Descrive un database come una collezione di asserzioni su un insieme di variabili, descrivendone vincoli e possibili combinazioni di valori
- Una volta definito il modello i dati archiviati è necessario seguire la struttura definita, senza possibilità di modifica

Object database:

- I dati vengono rappresentati come veri e propri oggetti, analogamente alla programmazione ad oggetti
- I database che sposano questo modello integrano anche un linguaggio di programmazione ad oggetti
- Molto comodi per le tecnologie web, data la trasparenza dei dati archiviati
- Molto gettonati per sistemi real-time

Document-oriented database:

- Sistema di archiviazione per informazioni document-oriented
- I documenti non hanno una struttura rigida, sono semi-strutturati
- Categoria principale dei database NoSQL[5]
- Le informazioni di un documento vengono archiviate in un oggetto in una singola istanza nel database
- Gli oggetti salvati possono essere tutti diversi fra loro
- I documenti sono equivalenti al concetto di oggetto nella programmazione

Stante il fatto che il progetto è stato condiviso con il collega Luca Sala, essendosi lui occupato di approfondire questa parte, lo studio di questi modelli di database verrà presentato più specificatamente nel suo elaborato.

In questa trattazione andremo ad analizzare solo il database più importante dal punto di vista progettuale ed il sistema più conosciuto sulla base del modello trattato.

2.2.1 Database document-oriented

Questa tipologia di software, come ogni database, è un sistema progettato per l'archiviazione, l'estrazione e la manipolazione di dati. In questo particolare caso, il termine "dati" è direttamente associato a documenti, nonché dati semi-strutturati.

Questa base di dati è una delle più diffuse della categoria NoSQL]: database basati su meccanismi differenti da quelli classici utilizzati nel modello relazionale.

I database document-oriented sono una sottoclasse dei database key-value, un paradigma di archiviazione dati che si occupa della gestione di strutture dati complesse, comunemente conosciute come tabelle hash. I dizionari, o tabelle hash, sono delle raccolte di record archiviate mediante l'utilizzo di una chiave, unica per ogni record, che verrà poi riutilizzata nella fase di ricerca del dato memorizzato all'interno del database per ritrovarlo ed estrarlo.

Le basi di dati orientate ai documenti stanno ad oggi contrastando i tradizionali database relazionali. La differenza maggiore è visibile nell'archiviazione dei dati: nei database relazionali, infatti, le informazioni vengono raccolte in tabelle, create dal programmatore ed i singoli oggetti vengono sparsi in più tabelle sulla base delle caratteristiche dell'oggetto. Al contrario, nei database document-oriented, gli oggetti da archiviare vengono memorizzati in una singola istanza nel database stesso, ed ogni oggetto può essere diverso da un altro.

Il fulcro di questa classe di database è la definizione di documento. Nonostante ogni sistema document-oriented ne distorca il significato vero e proprio, si presume che i documenti incapsulino dati codificati seguendo un formato standard. Infatti, i documenti che vengono generalmente memorizzati sono equivalenti alla struttura concettuale degli oggetti nella programmazione. La differenza risiede nel fatto che, in questo caso, non devono seguire uno schema definito, ma possono avere parti, sezioni e chiavi differenti. Un altro modo per descrivere un documento è pensarlo come un albero che può contenere diversi livelli di annidamento, esattamente come i suddetti documenti.

```
1  {
2      "Nome": "Matteo",
3      "Cognome": "Vanzini",
4      "Genere": "M",
5      "Corso_di_studio": "Scienze Informatiche"
6  }
```

Figura 2.2.1: generico documento json.

In figura 2.2.1 è possibile analizzare la struttura di un documento molto semplice, in cui troviamo quattro coppie <chiave:valore>, mentre in questo esempio non sono presenti dati annidati.

Un generico sistema di archiviazione dati orientato ai documenti, quando memorizza un documento, provvede ad associargli dei metadati [6] addizionali relativi al contenuto; questo per facilitare la ricerca del documento in un secondo momento, per motivi organizzativi o, ancora, legati alla sicurezza. Con metadati si intendono "dati su altri dati", cioè informazioni utili alla descrizione di altri dati. In questo caso vengono intesi i dati da archiviare.

I documenti vengono archiviati mediante una chiave univoca che li identifica: il campo "_id".

Le operazioni che un document-oriented database supporta, sono analoghe a quelle offerte da tutti gli altri sistemi, ma in questo caso sono definite come operazioni CURD. In particolare, l'acronimo è ricavato dai sostantivi inglesi Creation ("creazione" od "inserimento"), Retrieval ("interrogazione" o "ricerca"), Update ("aggiornamento" o "modifica") e Deletion ("eliminazione").

Per quanto riguarda l'operazione di interrogazione o ricerca (retrieval) i database di questo tipo offrono un API o un linguaggio di interrogazione che permette, come già accennato, di ritrovare i documenti sulla base del contenuto (metadati). Nonostante ciò, le prestazioni, come anche le opzioni di indicizzazione e configurazione disponibili, variano molto in base all'implementazione.

Per concludere è necessario parlare dell'organizzazione dei documenti. Vi sono differenti strade per strutturare un database document-oriented, quelle più comuni sono:

- Collection: raccolte di documenti, in cui, in base all'implementazione, un documento può essere memorizzato in una o più raccolte.
- Tags e metadati nascosti: dati aggiunti, inerenti al contenuto ma differenti.
- Cartelle gerarchiche: gruppi di documenti organizzati in una struttura ad albero.

Nella prossima sezione andremo ad analizzare una delle implementazioni di questo tipo di database, precisamente una tra le più diffuse e utilizzate: MongoDB.

2.2.2 MongoDB

MongoDB è un database NoSQL document-orented open-source, nato nel 2007 come servizio minore all'interno di un progetto più ampio, ma che, non molto tempo dopo, crescendo, è diventato indipendente. Il suddetto archivia documenti in formato JSON. I documenti sono raggruppati in collezioni (collection), ognuna delle quali può essere eterogenea o può non avere uno schema fisso. Le raccolte sono indipendenti, ovvero non vi sono relazioni o legami tra più raccolte.

Le caratteristiche fondamentali su cui MongoDB investe e che verranno qui di seguito analizzate, sono la scalabilità e l'alta disponibilità:

- *Scalabilità*:

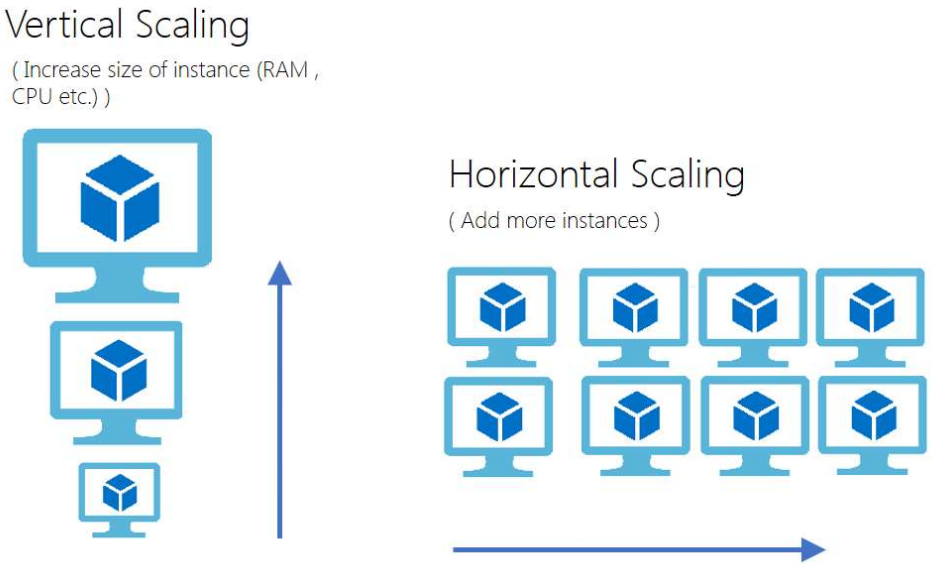

Figura 2.1: scalabilità verticale vs scalabilità orizzontale.

Esistono due tipi di scalabilità: quella verticale e quella orizzontale. MongoDB punta tutto sulla seconda, ovvero aumenta il numero di nodi per incrementare le capacità del sistema, riferendosi alla possibilità di realizzare un cluster. Grazie al meccanismo di sharding, infatti, è possibile distribuire le collezioni

in cluster di nodi, in modo da supportare grandi quantità di dati mantenendo performance elevate. Difatti, con sharding si intende la tecnica per la creazione di un sharded cluster, o database distribuito, in cui ogni nodo contiene una porzione del database.

- *Alta disponibilità*: consente di replicare i dati in modo molto più semplice rispetto ad altre soluzioni di storage. Questa pratica è utile nell'ottica di rafforzare l'affidabilità del sistema in caso di errori di qualunque genere. I due meccanismi che MongoDB utilizza sono Master-Slave e Replica Set. Quest'ultimo non è altro che una moltitudine di istanze contenenti i medesimi dati, dove è sempre presente un'unica istanza primaria e più istanze secondarie. Il funzionamento è semplice: quando il server primario riceve le nuove operazioni di scrittura, questo le esegue su sé stesso e poi invia le modifiche a tutti i server secondari in modo asincrono. Nel caso di errori del server primario, uno tra i secondari prenderà il suo posto e la catena verrà ristabilita.

Per utilizzare MongoDB è necessario installarlo in locale sulla propria macchina, ed una volta fatto è possibile interfacciarsi ad esso grazie ad una shell appositamente creata. Mediante quest'ultima è possibile effettuare, come di consueto, tutte le operazioni fondamentali, tra cui le operazioni CURD: creazione, interrogazione, modifica, eliminazione. Per cominciare da shell è possibile creare un database:

```
use <database>
```

Figura 2.2.2: creazione database.

Mediante il comando in figura 2.2.2 è possibile creare un nuovo database o riferirsi ad uno già precedentemente creato, come nel caso in figura 2.2.3, in cui prima viene selezionato il database da utilizzare e successivamente, specificando la collection su cui effettuare l'operazione, viene inserito un documento.

```
use myNewDatabase
db.myCollection.insertOne( { x: 1 } );
```

Figura 2.2.3: inserimento documento.

Per quanto riguarda l'operazione di aggiornamento di un documento (update), è possibile operare su un documento inserito mediante l'operazione "updateOne" per aggiornare un singolo documento, oppure, in alternativa, utilizzare "updateMany" per eseguire un aggiornamento in base ai criteri specificati su tutti i documenti presenti nella collezione.

Inoltre, è possibile ricercare i documenti in una collezione:

```
db.inventory.find( {} )
```

Figura 2.2.4: ricerca in una collection.

Eseguendo il comando mostrato in figura 2.2.4, la shell mostrerà tutti i documenti all'interno della collection "inventory".

Infine, è possibile trovare ed eliminare tutti i documenti di una collezione nel seguente modo:

```
db.inventory.deleteMany({})
```

Figura 2.2.5: eliminazione documenti.

Oltre a quelli riportati esistono molti altri comandi, dai più semplici a quelli maggiormente complessi. Per una documentazione accurata è sufficiente fare riferimento alla documentazione di MongoDB.

In generale, ad ogni operazione eseguita la shell restituirà una risposta. In particolare, la suddetta potrà essere fornita mediante la restituzione dell'output del comando eseguito o, in alternativa, mediante un messaggio il cui fine sarà quello di segnalare un possibile errore o un'esecuzione senza risultato.

Si prosegue ora con la presentazione di MongoDB presentando macroscopicamente l'indicizzazione che questo database offre.

Di default MongoDB crea un indice su campo singolo, in particolare sul campo "_id" di ogni collezione. La motivazione di questa scelta è presto detta: effettuare query in un database indicizzato è di gran lunga più veloce. Al contrario, senza indicizzazione, per poter trovare un riscontro, risulterebbe necessario scorrere ogni documento in ogni collezione, facendo così notevolmente aumentare il costo computazionale. Ad ogni modo, creare un indice su singolo campo, ma non solo, è molto semplice.

Verrà di seguito riportato un semplice esempio.

```
db.collection.createIndex( { name: -1 } )
```

Figura 2.2.5: creazione di un indice su singolo valore.

In figura 2.2.5 si può vedere come creare un indice sul campo "name". In questo caso si tratta di un indice in ordine discendente.

Per quanto riguarda la modellazione dei dati, invece, è necessario specificare che si tratta di una fase che, in tutti i database NoSQL, risulta molto delicata. Infatti, a differenza dei database relazionali in cui la progettazione segue delle fasi ben precise, in questo caso non esistono procedure studiate e definite in maniera consolidata, ma solo dei buoni consigli da seguire. La prima considerazione da fare è che, nel caso di MongoDB e di tutti i database document-oriented, i documenti sono schema-less, ossia senza schema: non esiste, dunque, uno schema fisso da seguire. Ciò significa che la modellazione si basa per lo più su scelte intuitive e di buon senso relative alla situazione. Quella sicuramente più importante riguarda le relazioni tra i documenti, in quanto in MongoDB non esistono vincoli di integrità referenziale. Le possibilità sono le seguenti:

- Documenti *incorporati*: questa scelta impone di incorporare i documenti collegati uno nell'altro.

```
1 {
2     "titolo": "La commedia",
3     "autori": [{ "nome": "Dante", "cognome": "Alighieri" }],
4     "edizioni": [
5         { "isbn": "xxxxxxx", "anno": 2014, "casaEditrice": "Edizioni X", "prezzo": 35.50, "ultima": true },
6         { "isbn": "yyyyyyy", "anno": 2013, "casaEditrice": "Edizioni X", "prezzo": 33.20 }
7     ]
8 }
```

Figura 2.2.6: esempio di documenti incorporati.

In figura 2.2.6 si nota un esempio di documento incorporato. Nello specifico sono stati incorporati due documenti: il primo con relazione molti a molti, ed il secondo con una relazione uno a molti.

Il vantaggio nell'eseguire questa procedura risiede nell'atomicità. Infatti, in MongoDB le uniche operazioni atomiche sono quelle che coinvolgono un singolo documento, ed in questo modo il suddetto limite viene rispettato. Un altro vantaggio è che, mantenendo tutto in un unico documento, avremo la garanzia che le modifiche avvengano senza salvataggi concorrenti, dunque senza errori. Infine, nel caso di sharding, per avere un documento di questo tipo non avremo bisogno di interrogare più nodi ma solo uno.

Si analizzino ora gli svantaggi. Anzitutto si presenta un possibile rischio di ridondanza, in quanto, se volessimo inserire un'opera dello stesso autore, le informazioni dell'autore stesso verrebbero duplicate. Conseguentemente, nel caso siano presenti dati ridondanti per i motivi di cui sopra, aggiornare i dati di un autore presente in più opere sarebbe molto costoso a livello computazionale.

- Documenti *referenziati*: le relazioni possono anche essere espresse mediante l'indicazione di una chiave, permettendo così a due documenti di essere collegati, in un processo molto simile all'utilizzo delle chiavi esterne nei database relazionali.

```
1  {
2      "titolo": "La commedia",
3      "autori": [ "dante" ],
4      "edizioni": [ "xxxxxxx", "yyyyyyy" ]
5  }
6
7  { "_id": "dante", "nome": "Dante", "cognome": "Alighieri" }
8
9  { "_id": "xxxxxxx", "anno": 2014, "casaEditrice": "Edizioni X", "prezzo": 35.50, "ultima": true },
10
11 { "_id": "yyyyyyy", "anno": 2013, "casaEditrice": "Edizioni X", "prezzo": 33.20 }
```

Figura 2.2.7: esempio di documenti referenziati.

In figura 2.2.7 vediamo come l'esempio di figura 2.2.6 diventerebbe in caso di documenti referenziati. In questo caso avrebbe ovviamente senso suddividere i documenti in tre collezioni: libri, autori e edizioni.

I vantaggi maggiori si hanno nel caso di relazioni molti a molti, infatti per modificare un autore collegato ad N opere dovremmo, in questo caso, eseguire una sola operazione. Analogamente, lo svantaggio maggiore risiede nel fatto che il numero di letture cresce all'aumentare delle referenze di un documento.

Come detto, non è quindi possibile stabilire delle regole generali per modellare i dati. Possiamo, tuttavia, tentare di seguire, con l'accortezza di sapere che la tecnica migliore potrebbe essere diversa caso per caso, una tra le pratiche appena presentate.

2.3 Framework

Dal momento che il database troverà la sua collocazione funzionale su un server, sarà necessario interfacciarsi ad esso mediante un'interfaccia web: lo strato di front end dell'architettura three-tier. La creazione di questo livello avviene, in generale, nello stesso momento in cui anche il livello di back end viene sviluppato. Nel caso di un applicazione web, come ora, per lo sviluppo ci si affida solitamente ad un framework.

Un framework è una struttura logica di supporto, il cui scopo è quello di semplificare il lavoro di sviluppo di un software ad un programmatore. Di fatto, è uno strumento che impone al programmatore una precisa metodologia di sviluppo.
In particolare, un framework è definito da un insieme di classi astratte e dalle relazioni che legano queste ultime tra loro. Alla base di un framework troviamo una serie di librerie di codice con uno o più linguaggi di programmazione ed una serie di strumenti di supporto allo sviluppo, tra cui un integrated development environment (IDE) ed un debugger. La sua funzione è quella di creare un'infrastruttura generale, lasciando al programmatore il compito di creare il contenuto vero e proprio. In questo modo, evita

allo sviluppatore la scrittura di codice già scritto precedentemente per strutture simili, lasciando che si concentri sulla parte più caratteristica dell'applicazione in sviluppo.
Tuttavia, esiste una sottocategoria di framework, che verrà di seguito trattata brevemente.
Un framework per applicazioni web è, a differenza di un generico framework come quelli di cui abbiamo parlato fino ad ora, un framework nato per supportare lo sviluppo di siti web dinamici e applicazioni web. L'obbiettivo primario del suddetto è, come detto prima, alleggerire il carico di lavoro di uno sviluppatore, evitandogli processi ripetitivi. Il principio fondamentale su cui un framework opera è riassunto dall'acronimo DRY: "don't repeat yourself". Diversamente da un generico framework, un web framework mette a disposizione altre strumentazioni: librerie per l'accesso a basi di dati, strumenti per la creazione di template HTML ed altri per la gestione delle sessioni dell'utente.
Infine, abbiamo un ulteriore sottocategoria dei framework per applicazioni web: i microframework. Questi ultimi, com'è intuibile dal nome, mancano di molte funzionalità che invece i framework per web application implementano. Una delle mancanze più rilevanti, sulla base delle specifiche tecniche precedentemente illustrate, è sicuramente l'astrazione database mediante object-relational mapping (ORM), ma come si vedrà tra poco, questo sarà un punto a favore della categoria di framework di cui stiamo parlando. Un ORM è una tecnica di programmazione che tenta di integrare software basati sulla programmazione ad oggetti con relational database management system (RDBMS), ovvero un DBMS basato sul modello relazionale. La suddetta tecnica, oltre ai notevoli vantaggi che porta, tra cui, il superamento dell'incompatibilità tra modello relazionale ed un progetto object-oriented, ed infine, l'elevata portabilità rispetto alla tecnologia DBMS usata, presenta anche un notevole svantaggio: l'utilizzo di un ORM rischia, in determinate situazioni, di compromettere le prestazioni del database vero e proprio.
È dunque stato affrontato uno studio per capire quale framework fosse più adatto alle esigenze di sviluppo del progetto.

La ricerca è stata mirata su un framework molto leggero, che lasciasse libertà sui vari componenti da utilizzare, come ad esempio il database, ma che fornisse un servizio efficace e affidabile.

Il primo framework preso in considerazione è stato Django, dato che è stato studiato approfonditamente nelle materie del corso di laurea. Quest'ultimo ha subito destato qualche dubbio a causa della gestione del database mediante ORM, in quanto, soprattutto nel caso in cui si fosse scelto un database NoSQL per lo sviluppo, come nel nostro caso, ne avrebbe ridotto le prestazioni e di conseguenza annullato i benefici.

Alla luce di queste considerazioni, si è optato, anche sulla base dell'analisi fatta nelle sezioni precedenti, per lo studio approfondito di un microframework abbastanza conosciuto: Flask.

2.3.1 Flask

Flask è un microframework web scritto in Python. Il suddetto è basato su WSGI [8], nonché l'interfaccia standard del web service per la programmazione in Python. Ancora, Flask utilizza Jinja2 [9] come motore grafico e possiede licenza BSD [10].

Questo microframework possiede un nucleo tanto scarno quanto solido. Non presenta, in qualità di microframework, alcuna astrazione per database, per la validazione dei form e molte altre componenti implementabili con librerie di terze parti.

Flask presenta una serie di caratteristiche. Verranno elencate quelle ritenute più importanti:

- Possiede un debugger ed un server per lo sviluppo
- Usa Jinja2 per i template
- Supporta i cookie di sicurezza (lato client)
- Basato su Unicode
- Compatibile con Google App Engine

La generica struttura di un applicazione web sviluppata in Flask presenta la seguente struttura:

```
AppName/
        static/
            css/
            js/
        templates/
            includes/
        app.py
```

Figura 2.2.8: struttura di un'applicazione Flask.

Verrà di seguito presentato il classico "Hello world", per dimostrare la versatilità e la semplicità dell'ambiente di sviluppo.

```python
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

Figura 2.2.9: "Hello World" Flask.

In figura 2.2.9 si può vedere, come anticipato, un'applicazione web a tutti gli effetti. La suddetta è una versione piuttosto scarna e priva di funzioni, ma si rivelerà utile per l'analisi del funzionamento. Il file da configurare è "app.py".

Come detto prima Flask è sviluppato in Python, di conseguenza le applicazioni web create mediante il suddetto è bene che seguano lo stesso linguaggio, per una migliore compatibilità.

Nella figura 2.2.9 è possibile notare un accenno di configurazione di "app.py". Infatti, configurando il suddetto file, ed eseguendolo sarà possibile collegarsi alla pagina "http://127.0.0.1:5000/" mediante un qualsiasi browser, ottenendo come risposta sul terminale dell'applicazione il messaggio "Hello, World!". Ciò è possibile per via della funzione "hello_world()" definita nel codice riportato in figura. Il decoratore python "app.route('/')" permette di abbinare la funzione "hello_world" con l'indirizzo "/". La conseguenza dell'abbinamento è che, come detto prima, una volta richiesto il collegamento alla pagina specificata, il sistema risponderà richiamando la funzione specificata, in questo caso "hello_world", che produce come output sul terminale la stringa "Hello, World!".

Una considerazione riguardo alla configurazione presentata: è possibile notare che per definire un "comportamento" è sufficiente specificare un percorso (grazie al decoratore "route") e la funzione da richiamare nel caso di collegamento a quell'indirizzo (in questo caso "hello_world"). Di conseguenza, è facile caprie quanto sia semplice costruire un albero di routing complesso e le relative funzioni.

Lo studio di Flask si concluderà con le ultime peculiarità di quest'ultimo.

Mediante semplici righe di codice da inserire all'interno del file "app.py", dopo l'inclusione delle librerie necessarie, è possibile utilizzare e configurare qualunque database ed integrarlo di conseguenza all'applicazione che si vuole sviluppare. Infine, senza l'utilizzo di librerie esterne, è possibile configurare l'indirizzo dell'applicazione e la porta hardware a cui collegarsi per usufruire del servizio.

In questa trattazione non sono state citate molte funzionalità offerte da Flask: è possibile consultare la documentazione completa per informazioni più dettagliate.

Parte II

Progetto MCIBase e algoritmo per la generazione dinamica di un Front End

Capitolo 3

Progettazione MCIBase

3.1 Introduzione alla progettazione

Questo capitolo si pone come obbiettivo la progettazione dell' applicazione web che soddisfi le specifiche precedentemente enunciate e quelle che verranno trattate dettagliatamente nella sezione 3.4.1.

3.2 Strategia progettuale

Come abbiamo visto nei capitoli precedenti, un'applicazione web si compone, sulla base dell'architettura three-tier, di tre livelli o strati, come rappresentato in figura 3.2.1.

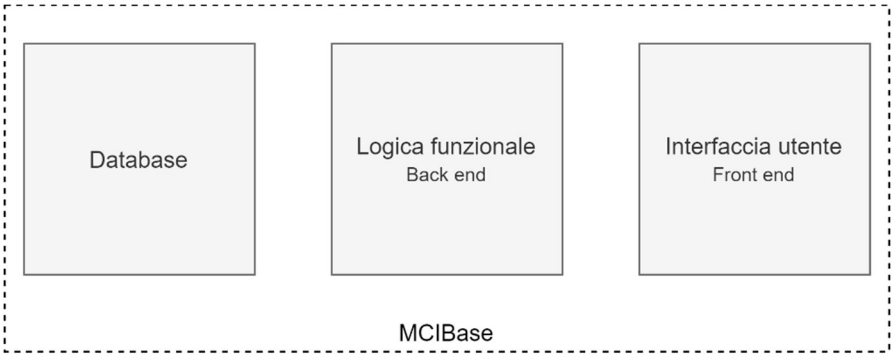

Figura 3.2.1: livelli dell'architettura three-tier da progettare.

Prendendo in considerazione l'analisi di MCIBase, nella sua interezza, è facile capire che la complessità è notevole; questo è il motivo per cui si è scelto di progettare i vari componenti dell'architettura singolarmente, uno alla volta. Nelle prossime sezioni andremo ad analizzare le fasi di progettazione di ogni livello, iniziando dal database, proseguendo con l'analisi della progettazione del back end e concludendo con la progettazione del front end dinamicamente generato.

3.3 Progettazione database

A causa della necessità di archiviare i dati si è reso necessario un database. Il primo step è stato quello della progettazione del suddetto, nonché lo strato dedicato alle operazioni di gestione e manipolazione dei dati. Successivamente alla fase dello studio tecnologico (capitolo due) è stata intrapresa una scelta,

valutando quale tecnologia fosse migliore sulla base delle specifiche tecniche più importanti, trattate nel primo capitolo.

La scelta fatta è stata ponderata sulla base delle richieste dell'equipe e delle specifiche progettuali. Per comprendere la scelta è di fondamentale importanza sapere che la ricerca, almeno per i primi tempi, sarebbe stata portata avanti in due sole città, Modena e Reggio Emilia, ma che in un secondo momento, avrebbe potuto essere estesa su territorio nazionale. Si è quindi ritenuto opportuno utilizzare un sistema potenzialmente distribuibile sulla rete nazionale, di conseguenza, su molteplici nodi della rete.

Un'altra motivazione per giustificare la decisione è che tra le specifiche ne era presente una tale per cui fossero memorizzabili dati con strutture diverse: essendo presenti diversi esami da fare, ognuno presentava una serie di dati da raccogliere diversi da quelli degli altri, arrivando quindi ad avere una diversa struttura per ogni esame. Successivamente è stata avanzata richiesta per la possibilità di inserire, in due determinati esami, due campi di dimensioni variabili: un vettore ed una matrice. I suddetti campi avrebbero potuto contenere dieci, cento, mille o più numeri, oppure anche essere vuoti, ma ad ogni modo non sarebbe stato possibile saperlo in anticipo. A causa di questa sequenza di richieste e necessità si è optato per l'utilizzo di MongoDB. Questo infatti, in quanto document-oriented database, archivia i dati in documenti senza schema, risolvendo il problema della lunghezza variabile dei campi e delle diverse strutture. Infine, grazie meccanismo di sharding e il favoritismo alla scalabilità orizzontale, MongoDB, si è rivelato perfetto per la possibilità di espandere la distribuzione su differenti nodi, nel caso la ricerca cresca a livello nazionale.

Prendiamo ora in considerazione una delle specifiche di cui abbiamo già parlato: la necessità di trattare dati provenienti da diversi esami, quindi con strutture diverse. Per una migliore suddivisione fisica, ma anche logica, dei documenti, s'è optato per la creazione di tante collezioni, una per ogni diversa struttura. Com'è stato spiegato nel capitolo precedente, nella sezione "MongoDB", l'omonimo database mette a disposizione le collection: raccolte

di documenti. Nel caso di questo progetto, come detto, le raccolte verranno organizzare sulla base delle strutture dati esistenti, creando di conseguenza tante raccolte quanti esami. Nella figura 3.3.1 si mostra una rappresentazione astratta di quella che sarà la suddivisione del database.

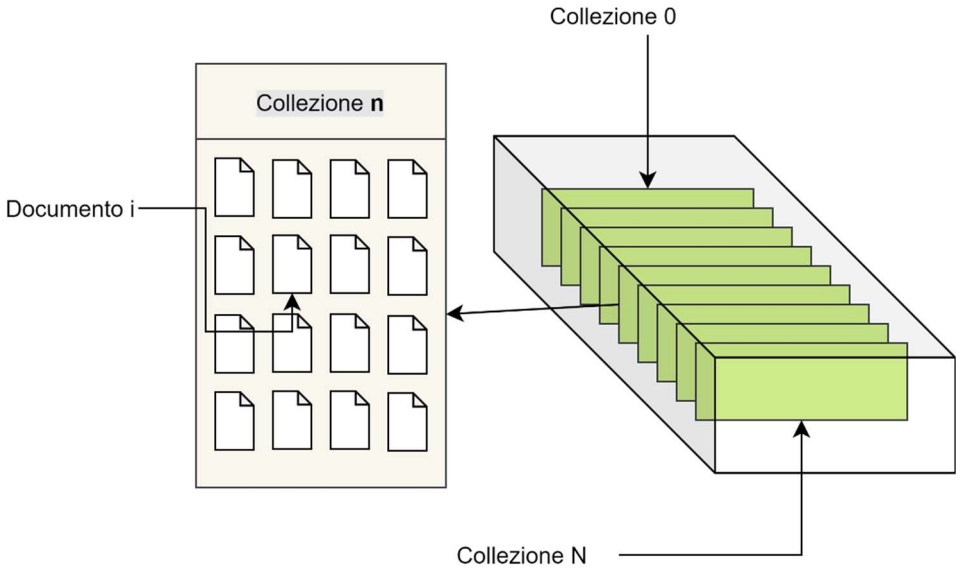

Figura 3.3.1: rappresentazione astratta di un database MongoDB (sulla destra) suddiviso in N collezioni, ognuna delle quali contiene I documenti (sulla sinistra).

A questo punto della trattazione è terminata la progettazione del primo livello: il database.

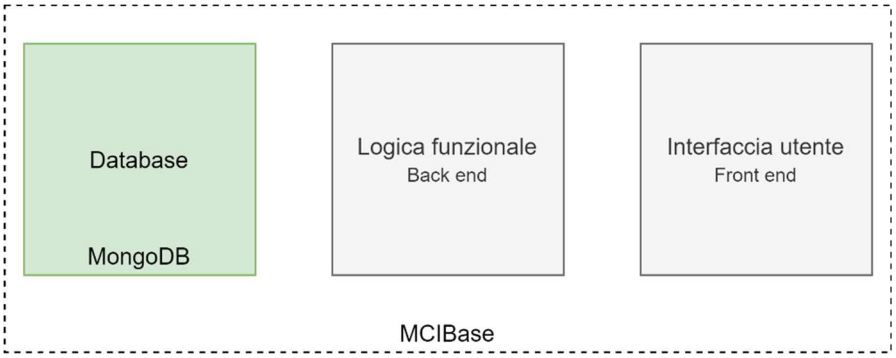

Figura 3.3.2: progettazione database terminata.

Nella prossima sezione sarà analizzata la progettazione del secondo strato.

3.4 Progettazione interfaccia di gestione

In questa sezione verrà trattata la progettazione dell'interfaccia di gestione.

Da un punto di vista teorico, come è stato mostrato nel primo capitolo, interfaccia utente e logica funzionale sono strati differenti, ma da un punto di vista pratico, vengono spesso sviluppati nello stesso momento. Infatti, risulterebbe difficile creare un back end completo senza conoscere con precisione la struttura dell'interfaccia utente, nonché il front end.

Sulla base delle richieste relative alla gestione dei dati e alle operazioni eseguibili per la manipolazione degli stessi sono state fatte considerazioni sulle tecnologie studiate nel secondo capitolo e di conseguenza è si è deciso di sviluppare l'applicazione web con Flask. Le motivazioni sono principalmente legate all'essenzialità della struttura del suddetto microframework e alla decisione di utilizzare MongoDB. Infatti, si è voluta evitare la compromissione delle prestazioni del suddetto database a causa dell'ORM intergato, come sarebbe stato, ad esempio, nel caso di Django. Inoltre, essendo Flask predisposto all'estensione mediante librerie di terze parti, è risultato perfetto per gli scopi progettuali.

Andremo successivamente a trattare le operazioni fondamentali che l'interfaccia dovrà essere in grado di supportare, per poi proseguire con la progettazione del back end e del front end.

3.4.1 Operazioni fondamentali

Tramite l'utilizzo dell'UI dovrà essere possibile eseguire le seguenti operazioni:

- Identificarsi come utente di una determinata categoria

- Inserire nuovi pazienti
- Inserire i dati degli esami di ogni paziente
- Visualizzare la lista dei pazienti
- Visualizzare i dati di un esame di un paziente
- Modificare i dati di un esame di un paziente
- Eliminare i dati di un esame di un paziente
- Scaricare tutti i dati di tutti i pazienti relativi ad un esame
- Dovrà essere possibile ordinare la lista dei pazienti in base a diversi criteri:
    - Alfabetico sul codice paziente
    - Data di inserimento del paziente nello studio
    - Esami svolti ed esami ancora da svolgere

Per far fronte a queste richieste si è deciso di progettare un'interfaccia con un'usabilità ben precisa, che verrà descritta dettagliatamente nella prossima sezione.

3.4.2 Navigabilità interfaccia

In questa sezione andremo, mediante l'utilizzo di un diagramma delle attività, a trattare le operazioni che possono essere effettuate mediante l'interfaccia che si vuole realizzare.

Prima di mostrare il diagramma vero e proprio è necessario trattare alcune regole secondo cui quest'ultimo si svolge.

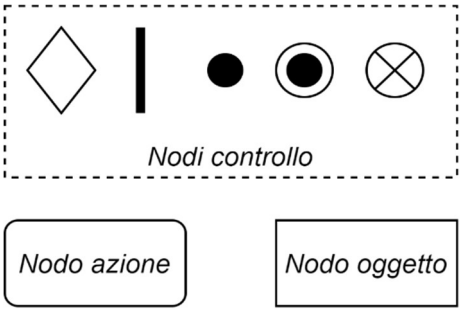

Figura 3.4.1: tipologie di nodi nei diagrammi di attività.

Come possiamo vedere nella figura 3.4.1, i diagrammi di attività si compongono di tre tipologie di nodi. Il primo che prenderemo in considerazione è il nodo azione, che rappresenta un'istantanea di comportamento. Ancora, abbiamo il nodo oggetto, nonché un nodo che rappresenta oggetti importanti, come input e output di determinate azioni. Infine, abbiamo i nodi controllo, che sono necessari per descrivere il flusso delle attività.

Dopo aver dato le nozioni base per comprendere uno schema è possibile procede con l'analisi dello stesso.

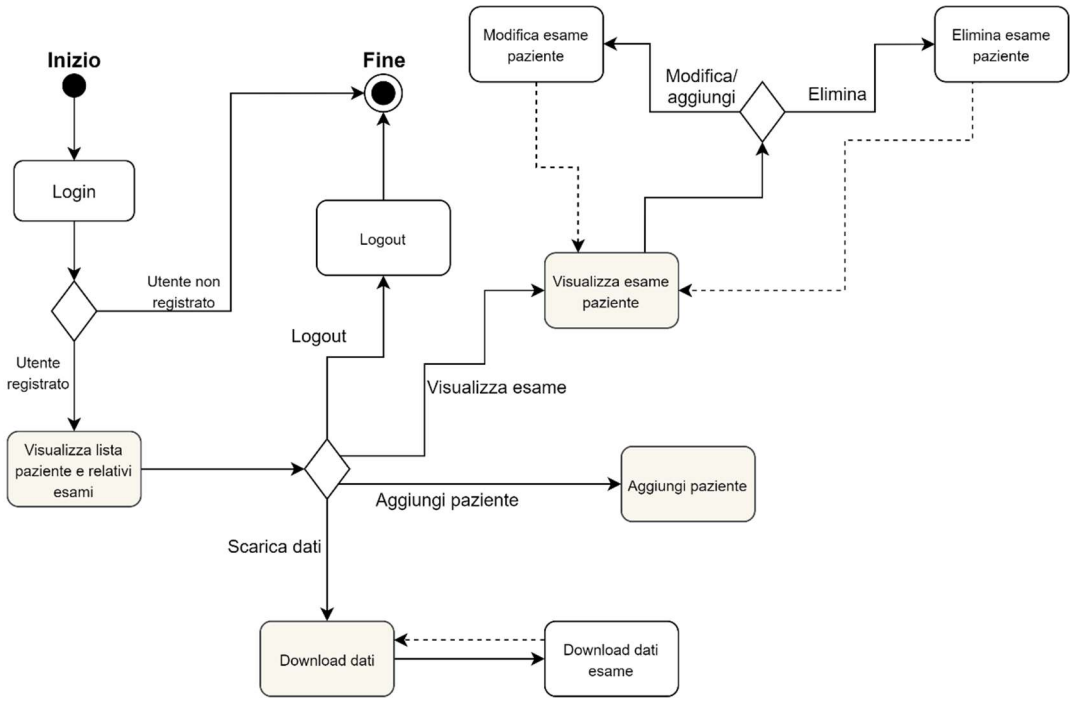

Figura 3.4.2: diagramma di attività MCIBase.

Com'è possibile notare in figura 3.4.2, una volta effettuato il login ci si ritroverà nella Home del sito, nonché la pagina che conterrà la lista dei pazienti ed i relativi dati. Successivamente sarà possibile muoversi in diverse direzioni. L'utente autenticato potrà visualizzare i dati relativi ad un esame di un paziente della lista, per poi aggiungerne, modificarli o rimuoverli. Infine, potrà inserire un nuovo utente o, nel caso sia necessario, scaricare i dati cumulativi di tutti i pazienti relativi ad un esame in particolare.

3.4.3 Gestione utenti

In questa sezione verrà trattata la gestione degli utenti. Dal momento che tra le richieste ne era presente una per cui gli utenti fossero categorizzati si è pensato, sulla base delle direttive ricevute dall'equipe, di creare le seguenti categorie di utente: amministratore, super-ricercatore, ricercatore e studente.

La prima categoria, come dal nome, si occuperà di gestire l'applicazione: sarà in grado di manipolare gli utenti esistenti e di crearne dei nuovi. La seconda categoria, il super-ricercatore, avrà invece la possibilità aggiungere, rimuovere e modificare tutti i dati presenti sul database, con la possibilità di scaricare ogni dato desiderato. Diversamente, i ricercatori, potranno aggiungere esami ed effettuare il download, ma potranno modificare ed eliminare solo i dati da loro stessi inseriti. L'ultima categoria invece, lo studente, potrà solamente scaricare i dati dalla piattaforma, il resto delle operazioni gli saranno negate.

La categorizzazione appena trattata è stata realizzata per una questione di sicurezza. La categoria studente, per esempio, è stata introdotta per gli eventuali studenti che parteciperanno allo studio, per consentirgli l'elaborazione dei dati, ma per evitare problemi legati all'inserimento errato di dati o l'erronea eliminazione/modifica di quelli già presenti.

3.4.4 Controllo dati

A questo punto è necessario pensare alla progettazione di un sistema che limiti il più possibile l'errore umano. Questo è il motivo principale per cui, a seguito della richiesta tale per cui un documento inserito venga completato automaticamente sulla base di vincoli precedentemente specificati, si è progettato un sistema in grado di adempiere a questi compiti. Il suddetto sistema, una volta implementato, entrerà in gioco durante la fase di inserimento di un documento. In particolare, è bene specificare che prima che tale sistema

possa funzionare sarà necessario specificare, nelle strutture dei diversi esami, quali campi dovranno essere inseriti automaticamente in questa fase. Una volta dichiarati i vincoli, il sistema sarà in grado di aggiungere al documento da inserire i campi specificati, rendendo l'inserimento più sicuro e meno soggetto ad errori.

In figura 3.4.3 verrà rappresentato in modo grafico la differenza tra questo sistema e quello classico.

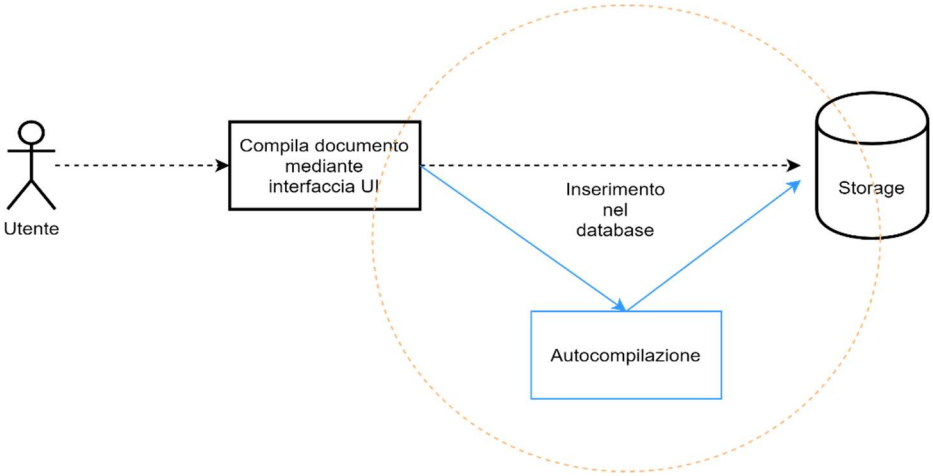

Figura 3.4.3: sistema controllo dati.

Alla fine di questa fase, si può dire completata la progettazione del back end.

Si procederà nella prossima sezione con l'analisi dell'interfaccia utente, il livello di front end.

3.5 Progettazione algoritmo front-end

Nella seguente sezione verrà affrontata la progettazione del terzo strato dell'architettura three-tier: l'interfaccia utente. Per riuscire a creare un UI funzionale ed efficiente si è, ovviamente, tenuto conto, in primis, delle specifiche presentate dall'equipe, ed infine delle regole classiche per l'usabilità delle applicazioni web: semplicità ed efficienza. Dal momento che per definizione lo strato di logica funzionale deve collegare l'interfaccia al

database, è facile affermare che le specifiche su cui, sia il back end che il front end sono fondati, sono le medesime. Per questo motivo si è deciso di sviluppare l'interfaccia sulla base delle scelte intraprese nella progettazione del back end.

Proseguiremo con un accurata analisi, per capire quali e quante pagine sono necessarie per un'interfaccia completa.

Le operazioni di inserimento, visualizzazione e modifica richiedo una pagina ognuno. Ne richiede un'altra l'operazione di visualizzazione della lista dei pazienti. Ancora, ne è necessaria una ulteriore per poter mostrare gli esami di cui poter scaricare i dati collezionati, ed infine, una per il login. Per l'operazione di eliminazione non è necessaria alcuna pagina, dal momento che un bottone apposito potrà essere posizionato nella pagina di visualizzazione. Analizzando ora il database su cui si è deciso di sviluppare il progetto, è possibile iniziare a ipotizzare delle soluzioni per la questione relativa alle diverse strutture degli esami. Tratteremo di seguito due diversi approcci alla realizzazione dell'interfaccia, sulla base delle considerazioni fino ad ora fatte.

In primis prendiamo in considerazione un approccio statico, che costringerebbe il programmatore a creare, date le diverse strutture, tante pagine di inserimento quante sono le diverse strutture, ed applicando lo stesso ragionamento per quanto riguarda le operazioni di visualizzazione e modifica è facile capire che diventerebbe necessario creare decine di pagine HTML molto simili, ma differenti. Questo è il principale motivo per cui non si è cercata un'altra soluzione.

L'alternativa, infatti, è stato l'approccio dinamico, cioè la creazione di un'interfaccia generica adattabile alle diverse strutture. Si è pensato, in particolare, di creare una pagina HTML standard, che avrebbe poi integrato una parte esterna creata dinamicamente. Il componente dinamico sarebbe stato creato sulla base della struttura dell'esame selezionato, per poi essere integrato ad una pagina standard statica.

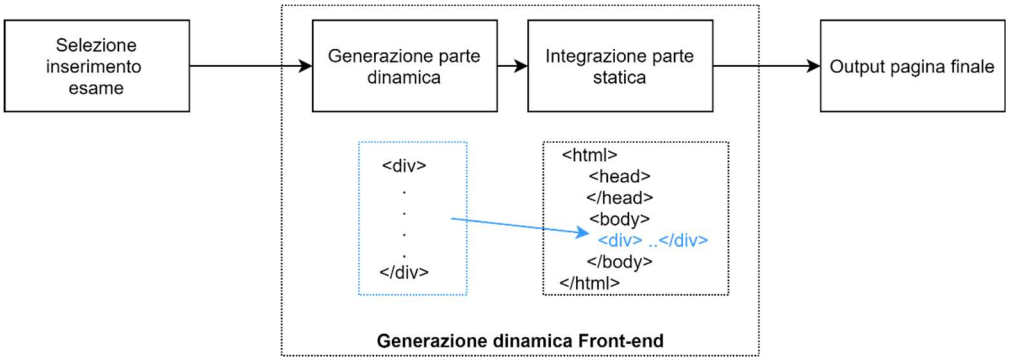

Figura 3.5.1: processo creazione di una pagina HTML dinamica.

Come è possibile vedere in figura 3.5.1, una volta selezionato l'esame da visualizzare o inserire, verrà generata una porzione di codice HTML da integrare ad una pagina HTML statica. L'insieme dei due componenti andrà a comporre la pagina finale da mostrare all'utente.

La conseguenza della generazione dinamica è la possibilità di creare una sola pagina di inserimento ed una di visualizzazione.

Il componente che si occupa della generazione della porzione di codice HTML è il file "html_generator.py", che rappresenta l'implementazione dell'algoritmo vero e proprio, che verrà trattato in modo approfondito nel capitolo successivo: l'implementazione dell'applicazione web.

3.5.1 Tolleranza agli schemi

Come anticipato nella sezione precedente, l'utilizzo di un approccio dinamico è stato fondamentale. Questa scelta è stata dettata, oltre che per una questione di comodità del programmatore, per motivi legati alle richieste iniziali dell'equipe. Tra le richieste, infatti, ne era presente una tale per cui fosse possibile, durante lo sviluppo dell'applicazione stessa, aggiungere o rimuovere campi dalla struttura degli esami. Di conseguenza, con un approccio statico sarebbe stato necessario, nel caso di modifiche, alterare manualmente ogni file contenente la struttura modificata. In questo modo, invece, generando le pagine

sulla base della struttura modificata, non è più necessario preoccuparsi di modificare i file a cascata, perché questi verranno generati con le modifiche già apportate alla pagina.

Il sistema di generazione dinamica ha portato, infatti, ad una tolleranza agli schemi tale da poter modificare le strutture in qualunque momento, senza ripercussioni sull'interfaccia "vecchia".

Finita la progettazione di questo livello, si conclude la progettazione di MCIBase.

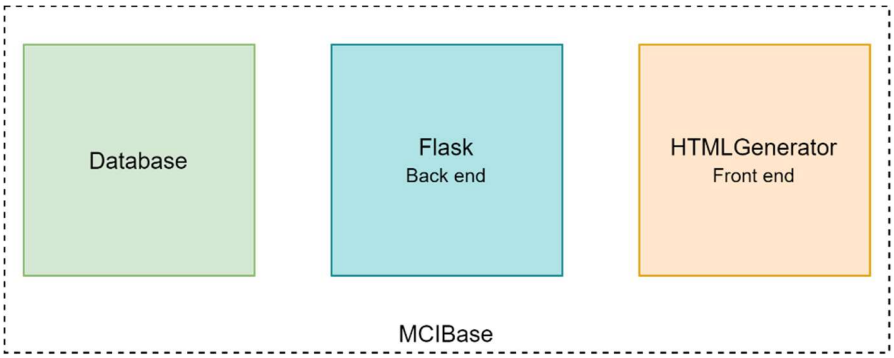

Figura 3.5.2: step finale della progettazione.

Come si può vedere in figura 3.5.2, a progettazione terminata, avremo i tre strati dell'architettura three-tier realizzati mediante, in odine dal primo livello, MongoDB, Flask ed infine "html_generator".

Capitolo 4
Implementazione MCIBase

4.1 Introduzione all'implementazione

In questo capitolo si andrà a trattare l'implementazione di alcuni dei livelli dell'architettura di cui abbiamo parlato nel capitolo precedente. Inizialmente si analizzerà la struttura generale dell'applicazione e l'integrazione di MongoDB con Flask, per poi andare ad analizzare nello specifico l'algoritmo di generazione dinamica del front end ed alcuni altri punti importanti dell'applicazione, ad esempio il download dati. Per trattare accuratamente la fase implementativa verranno fatti alcuni riferimenti

al capitolo precedente, andando a collegare le fasi della progettazione alla realizzazione vera e propria dei componenti di MCIBase.

4.2 Sguardo generale

MCIBase è strutturalmente uguale alle più classiche applicazioni web sviluppate con Flask.

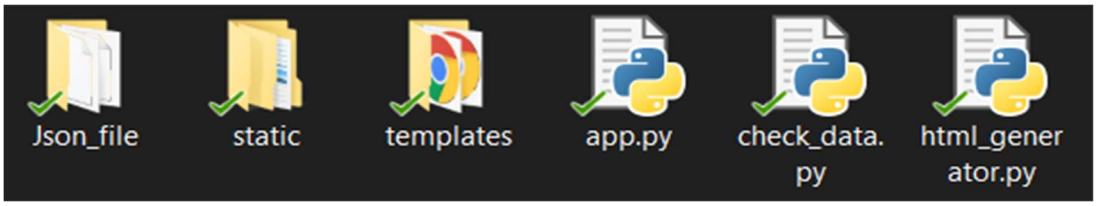

Figura 4.2.1: struttura applicazione MCIBase.

Il file principale, "app.py", include alcune librerie come ad esempio "PyMongo", per l'interazione col database. Inoltre, nel medesimo file vengo inclusi anche altri due file. I suddetti rappresentano l'implementazione di parti ben specifiche del progetto: "check_data.py" e "html_generator.py". Questi due file rappresentano rispettivamente l'implementazione del controllo dati e l'algoritmo di generazione dinamica del front end. Come di consueto, nelle cartelle "static/css/" e "static/js/" sono presenti, rispettivamente, i fogli di stile relativi alle pagine HTML e gli script Javascript per l'implementazione di determinate funzioni grafiche del front end. Ancora, nella cartella "templates/" sono presenti tutte le pagine HTML dell'applicazione web, comprese le pagine che contengono solo uno scheletro vuoto, come ad esempio, inserimento e visualizzazione. Infine, è presente un'ultima cartella: "Json_file/". In quest'ultima cartella sono stati inseriti tutti i file JSON corrispondenti alle diverse strutture degli esami. Questi sarebbero dovuti essere utilizzati come file di validazione in MongoDB, ma di fatto, sono stati utilizzati nei due file di cui si è parlato poco fa, per generare il front end e per controllare i dati nella fase di inserimento.

4.3 App.py

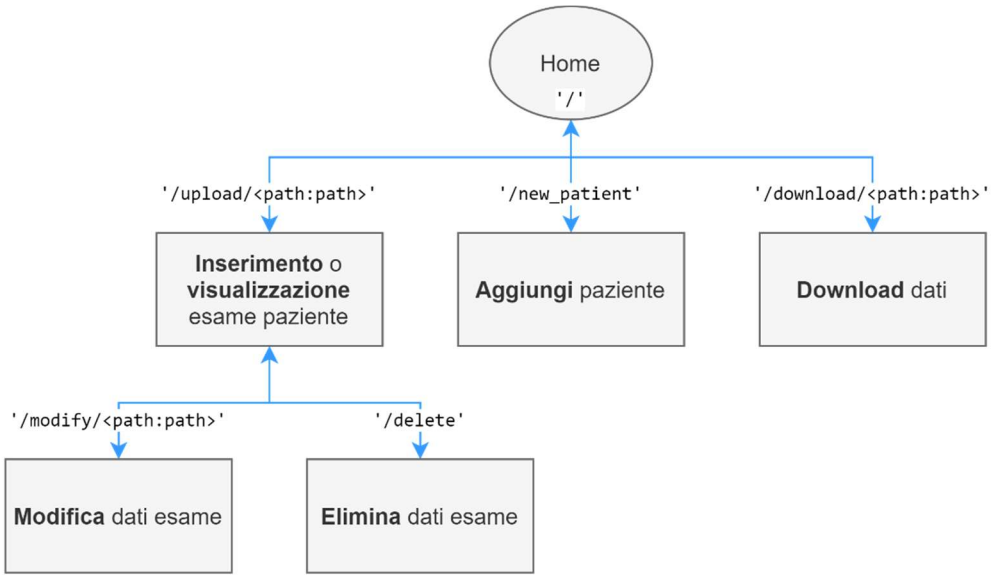

Figura 4.2.2: albero di routing interno di MCIBase.

Si noti in figura 4.2.2 l'albero delle possibili operazioni eseguibili mediante MCIBase. Ogni operazione è raggiungibile dalla home del sito, nonché la pagina contenente la lista di pazienti ed i relativi esami.

Si mostrerà ora, all'interno del file "app.py", come l'albero di figura 4.2.2 è stato realizzato. Sulla base di ciò che è stato detto nel secondo capitolo, lo studio delle tecnologie, configurare Flask significa anche associare ad un decoratore una funzione "azione". Infatti, nell'immagine 4.2.3, ovvero il file "app.py" è possibile notare come ad ogni "@app.route" segue sempre una funzione, nonché la funzione associata. Nella medesima figura è possibile notare anche che prima della dichiarazione dell'albero di routing dell'applicazione avviene un'operazione fondamentale per la pre-configurazione del sistema: la connessione al database.

```
15  def create_app():
16
17      app = Flask(__name__)
18      csrf = CSRFProtect(app)
19      csrf.init_app(app)
20      #specifica url db
21      app.config["MONGO_URI"] = "mongodb://localhost:27017/BioBase"
22      app.config['SECRET_KEY'] = [redacted]
23      #da qui in poi useremo mongo.db per far riferimento al nostro db
24      mongo = PyMongo(app)
25
26      #generazione collection se inesistenti
27      exams = ['nps0', 'nps1', 'epic', 'ls', 'blood', 'csf', 'pet', 'mri0', 'mri1', 'clinics', 'demographi
28
29      extra_collection = ['patient','users']
30      list_of_existing_collection = mongo.db.list_collection_names()
31 >    for e in exams+extra_collection: ···
34
35
36      @app.route('/bar', methods=['GET', 'POST'])
37 >    def change_features(): ···
40
41      @app.route('/statistics', methods=['POST','GET'])
42 >    def statistics(): ···
74
75      @app.route('/download/<path:path>', methods=['POST','GET'])
76 >    def download(path = None): ···
134
135     @app.route('/select/<path:path>', methods=['POST','GET'])
136 >   def select(path = None):  ···
144
145     @app.route('/login', methods=['POST', 'GET'])
146 >   def login(): ···
168
169     @app.route('/registration', methods=['POST', 'GET'])
170 >   def registration(): ···
201
202     @app.route('/delete',  methods=['GET','POST'])
203 >   def delete(): ···
216
217     @app.route('/',  methods=['GET','POST'])
218 >   def index(): ···
252
253     @app.route('/new_patient',  methods=['GET','POST'])
254 >   def new_patient(): ···
294
295     @app.route('/modify/<path:path>',  methods=['GET','POST'])
296 >   def modify(path = None): ···
315
316     # @login_required
317     @app.route('/upload/<path:path>', methods=['GET','POST'])
318 >   def upload(path = None): ···
349
350     @app.route('/check_data/<path:path>', methods=['GET','POST'])
351 >   def check_data(path = None): ···
470
471     @app.route('/admin/remove', methods=['GET','POST'])
472 >   def admin_remove_all(): ···
477
478     @app.errorhandler(CSRFError)
479 >   def csrf_error(reason): ···
481
482     return app
483
484 > if __name__ == '__main__': ···
488
```

Figura 4.2.3: app.py .

Si noti che la figura 4.2.3 non riporta il codice del file “app.py” finale, bensì una versione precedente.

4.4 Libreria PyMongo

Come detto nella sezione precedente e com'è possibile vedere nella figura 4.2.3, all'interno del file "app.py" è stata inclusa la libreria PyMongo. Di seguito andremo richiamare qualche elemento di teoria sulla suddetta API, per poi analizzarne l'utilizzo.

PyMongo è il driver ufficiale per l'interfacciamento con MongoDB da linguaggio Python. Il suddetto permette, dopo averlo installato, di configurare ogni aspetto della connessione al database. In particolare, permette di specificare l'indirizzo su cui risiede il database (per potersi connettere), la porta tramite cui accedervi ed infine permette di creare un'istanza del database, in questo caso denominata "mongo" (figura 4.4.1), mediante il quale eseguire le operazioni sui componenti del database, tra cui inserimento, eliminazione e ricerca.

```
mongo = PyMongo(app)
```

Figura 4.4.1: istanza MongoDB

In figura 4.4.1 si nota come alla variabile "mongo" venga assegnata l'istanza dell'oggetto "PyMongo", che, come detto prima, rappresenta il nostro database MongoDB. Mediante la suddetta variabile vengono eseguite tutte le operazioni sul database: ne verrà riportata una a scopo illustrativo.

```
mongo.db[path].insert(document,check_keys=False)
```

Figura 4.4.2: esempio di inserimento nel database.

Nell'esempio riportato in figura 4.4.2, viene inserito il documento "document" nel database, per precisione, nella collection il cui nome corrisponde alla stringa contenuta nella variabile "path".

Perché PyMongo funzioni correttamente, però, è necessario prima di tutto, installare MongoDB. L'installazione di MongoDB, su sistema operativo Windows (usato per lo sviluppo del progetto), è molto semplice: è necessario scaricare dal sito ufficiale di MongoDB l'installer e lanciarlo. Una volta terminata questa operazione sarà possibile

eseguire il servizio MongoDB e successivamente interfacciarsi ad esso mediante l'API PyMongo.

4.5 Implementazione HTMLGenerator

In questa sezione verrà trattata la parte relativa all'implementazione dell'algoritmo per la generazione dinamica del front end. In primis, verrà trattata la struttura dei file di validazione, nonché una parte fondamentale dell'algoritmo stesso, in secundis, si presenterà l'idea astratta dell'algoritmo, in terzis, verrà mostrato come dall'idea si è passati alla realizzazione, ed infine, verranno commentate le righe di codice ritenute più importanti dell'algoritmo.

4.5.1 Validation file

Come anticipato in questa sezione verranno trattati i file di validazione. In particolare, verrà descritta dettagliatamente la struttura di uno dei suddetti file. Questa analisi sarà di fondamentale importanza per capire la realizzazione dell'algoritmo, dal momento che lo sviluppo del suddetto ha come unica e fondamentale base tale struttura.

Prima dell'analisi vera e propria del file di validazione è necessario sapere cos'è e perché viene utilizzato. Un file di validazione è un semplice file JSON, nonché un documento, che però in questo caso non conterrà informazioni relative ad un oggetto da memorizzare (come ad esempio le generalità di una persona), bensì i vincoli che tali informazioni devono rispettare. Verrà riportato un esempio.

```
{
    "$jsonSchema" : {
        "bsonType": "object",
        "title": "Persona",
        "required": [ "nome", "cognome" ],
        "properties": {
            "nome": { "bsonType": "string" },
            "cognome": { "bsonType": "string" },
            "data_di_nascita": { "bsonType": "date" },
            "codice_fiscale": { "bsonType": "string" },
            "genere": {
                "bsonType": "string" ,
                "enum": ["M", "F", "Altro"]
            }
        }
    }
}
```

Figura 4.5.1: validation file esempio.

```
{
    "nome": "Matteo",
    "cognome": "Vanzini",
    "data_di_nascita": "1997-11-27",
    "codice_fiscale": "MTTVZN97S27F257R",
    "genere": "M"
}
```

Figura 4.5.2: esempio di documento json.

In figura 4.5.1 è riportato un file di validazione che, in questo caso, vincola le caratteristiche delle generalità di una persona. Allo stesso modo, in figura 4.5.2 è riportato un documento che rispetta i vincoli applicati dal file di validazione della figura 4.5.1 . In generale, un file di validazione serve a specificare i vincoli e le restrizioni che determinati documenti devono rispettare per poter essere inseriti nel database. Nello specifico, è bene specificare che ad ogni collezione può essere associato uno ed un solo file di validazione. Nel caso di questo progetto, infatti, l'obbiettivo era quello di creare un file di validazione per ogni esame, controllando infine che i documenti inseriti nelle diverse collection progettate rispettassero le strutture imposte. Inoltre, un file di validazione, può contenere diversi vincoli, anche annidati tra loro. Questo avviene perché, come possiamo vedere in figura 4.5.1, ad un determinato "bsonType", nonché una parola chiave a cui corrisponde la tipologia del dato che vogliamo venga inserito (number, string, date, object o array), sono direttamente collegate altre parole chiave, nonché le proprietà di quel tipo di campo. In particolare, molte delle proprietà legate al "bsonType" "object", saranno diverse da

quelle legate ad “array”, come da quelle di ogni altro “bsonType”. Tuttavia, le regole di scrittura di questi file ci permettono di inserire un vincolo dentro ad un altro. Prendiamo in considerazione l’esempio in figura 4.5.1. Il documento stesso, come si può vedere, ha il “bsonType” che corrisponde a “object” e nelle sue proprietà sono elencati altri oggetti con differenti “bsonType”. Questo è un esempio di documenti annidati: oggetti che contengono altri oggetti.

In figura 4.5.3 è possibile vedere quali sono le generiche proprietà associate ai “bsonType” più utilizzati all’interno dei validation file di MCIBase.

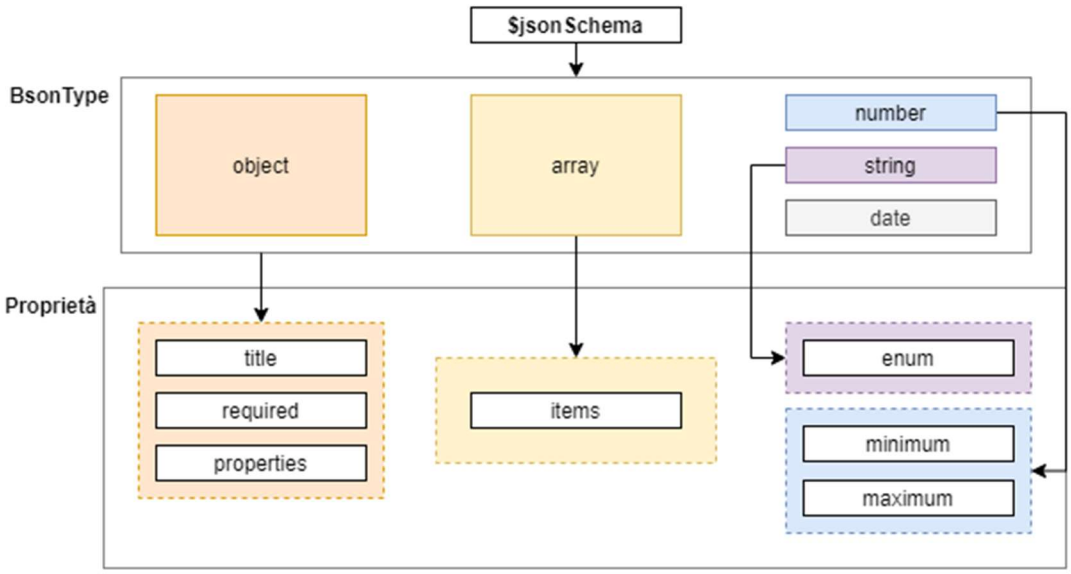

Figura 4.5.3: bsonType dei file di validazione.

4.5.2 Pseudocodice

Terminata la descrizione di un file di validazione e chiarita la generica struttura è possibile iniziare a parlare di pseudocodice, utile per capire a grandi linee le funzioni che caratterizzeranno la struttura dell’algoritmo finale.

```
esplora(esame):

    schema = readFile("jsonFile/"+esame+".json")
    if jsonSchema in schema.keys():
        schema = schema["$jsonSchema"]
    scheletro = {}
    smista(schema, scheletro)
```

Figura 4.5.4: pseudocodice funzione esplora.

```
smista(schema, scheletro, array = False):

    type = schema[bsonType]
    if type == object:
        for p in properties:
            schema = schema[p]
            smista(schema, scheletro)
    if type == array:
        for i in items:
            schema = schema[i]
            smista(schema, scheletro, array = True)
    altrimenti:
        converti(schema, scheletro, array)
```

Figura 4.5.5: pseudocodice funzione smista.

```
converti(schema, scheletro, array):

    type = schema[bsonType]
    if type == date:
        scheletro.append("<input type='date'>")
    if type == numeric:
        scheletro.append("<input type='number' minumim=" " maximum=" ">")
    if type == string:
        if array:
            for e in enum:
                scheletro.append("<input type="checkbox" value="+e+">")
        if enum in schema:
            scheletro.append("<select>")
            for e in enum:
                scheletro.append("<option value="+e+">")
            scheletro.append("</select>")
        altrimenti:
            scheletro.append("<input type="text">")
```

Figura 4.5.6: pseudocodice funzione converti.

Nelle figure 4.5.4, 4.5.5 e 4.5.6 è riportato lo pseudocodice delle tre funzioni che comporranno l'algoritmo. Si andrà ora ad analizzare brevemente il codice, per capirne il funzionamento.

La prima funzione è "esplora": si occupa di leggere lo schema di un esame dal relativo file JSON, di controllare che il file letto sia effettivamente un file di validazione, di creare la lista che conterrà il codice HTML ed infine di richiamare la funzione "smista", che accetta come argomento posizionale lo schema caricato e lo scheletro vuoto. La seconda funzione invece si occupa, come intuibile dal nome, di smistare lo schema ricevuto come argomento, sulla base del primo bsonType trovato al suo interno. La prima operazione che esegue, infatti, è l'assegnamento del bsonType alla variabile "type" che verrà successivamente utilizzata per capire come gestire lo schema che si sta analizzando. Ad ogni modo, a seguito delle operazioni eseguite in base al bsonType, la funzione "smista" può terminare in due modi: richiamando sé stessa in modo ricorsivo o richiamando la funzione "converti" passandole come argomenti posizionali lo schema, lo scheletro ed una variabile di tipo booleano di nome "array". La terza ed ultima funzione, sulla base del bsonType dello schema ricevuto, aggiunge allo scheletro una porzione di codice HTML di diverso tipo.

4.5.3 Html_generator

Una volta chiarita la struttura generale di un file di validazione e l'idea alla base di html_generator è possibile analizzarne l'effettiva implementazione. L'algoritmo, infatti, una volta selezionata la pagina dell'esame e l'operazione che si vuole eseguire (inserimento, modifica o visualizzazione), avrebbe dovuto scansionare il relativo file di validazione ed infine generare il codice HTML corrispondente. Per capire cosa si intende con "codice HTML corrispondente" è necessario suddividere i "bsonType" utilizzati in due gruppi:

- Stiring, Number, Date
- Array, Object

Il primo gruppo rappresenta gli elementi semplici, ovvero gli elementi che possono essere convertiti direttamente in semplici "input" HTML (figura 4.5.7).

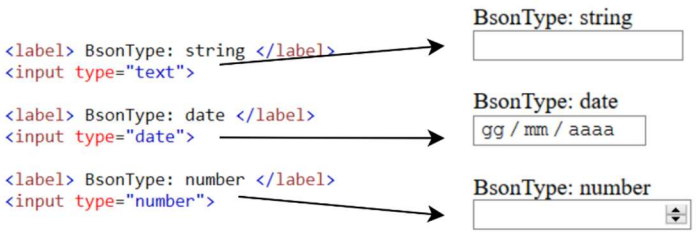

Figura 4.5.7: conversione elementi semplici.

Il secondo ed il terzo gruppo, invece, rappresentano un insieme di oggetti i quali possono avere qualunque "bsonType" della lista. La differenza tra "Array" e "Object" è data dal fatto che, nel primo tipo, gli oggetti nella lista devo avere tutti lo stesso "bsonType", mentre nel secondo, "Object", possono anche essere tutti diversi. La difficolta nella conversione in elementi HTML sta nel fatto che tutti gli elementi di un "Array", ad esempio, devono diventare una lista di elementi "input" di tipo "checkbox" contenuti in un "div", mentre gli Object, in qualità di "contenitori" sono dei semplici "div" che andranno poi a contenere molteplici elementi di diverso tipo. Di seguito verranno riportati degli esempi, uno per ogni "bsonType" del secondo gruppo.

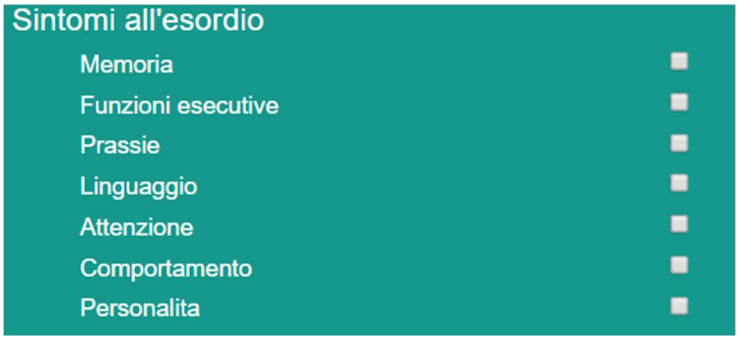

Figura 4.5.8: conversione HTML di un oggetto di tipo Array.

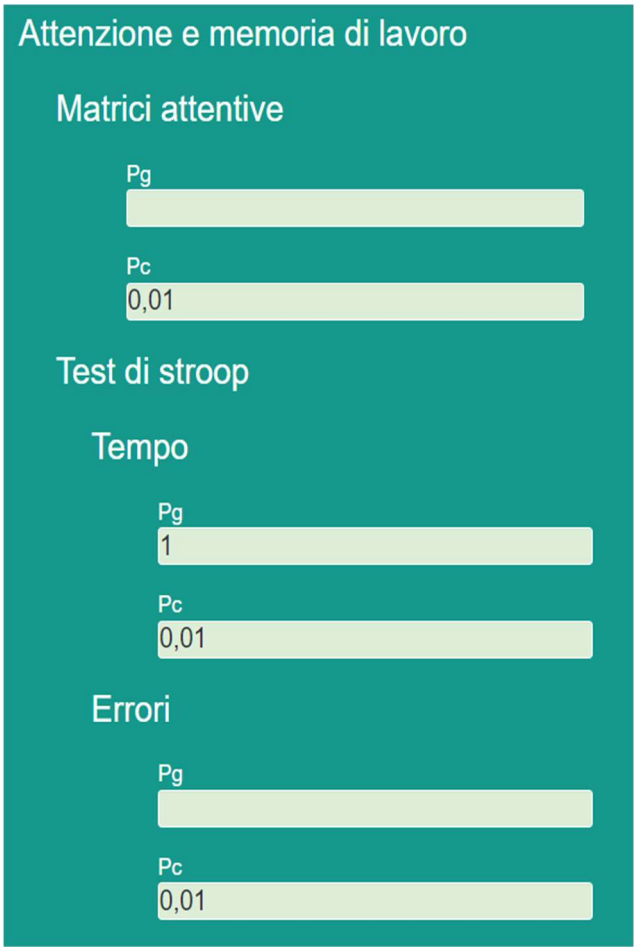

Figura 4.5.9: conversione HTML di un oggetto di tipo Object.

Chiarita la parte più teorica è possibile analizzare il funzionamento dell'algoritmo. Html_generator entra in gioco quando l'utente seleziona un esame di un paziente nella homepage. In quel momento il back end di MCIBase si occuperà di capire se mostrare la pagina di visualizzazione, qualora i dati fossero già stati inseriti precedentemente,

oppure, in caso contrario, mostrare il form di inserimento. Ovviamente, come già detto e ripetuto, la peculiarità dell'applicazione è che, ad entrambi le operazioni suddette, il sistema risponderà dinamicamente, generando sul momento la pagina desiderata. Si andranno ad analizzare i due differenti comportamenti: l'inserimento e la visualizzazione dei dati.

Quando l'utente richiede l'inserimento dei dati relativi ad un esame il sistema risponde richiamando la funzione "update" di "app.py". La suddetta funzione, dopo dei controlli di sicurezza relativi al percorso, all'autenticazione dell'utente e al tipo di connessione utilizzata per giungere alla pagina, richiama la funzione "explore" (corrispondente a "esplora" nello pseudocdice) di "html_generator.py", che si occuperà, insieme ad altre funzioni, della creazione dell'HTML della pagina richiesta. La funzione "explore", precisamente, dopo aver recuperato lo schema dell'esame dal file JSON che viene istanziato in un dizionario Python, rimanda la gestione ad un'altra funzione: "switch" (corrispondente a "smista" nello pseudocodice). Quest'ultima funzione è il fulcro dell'algoritmo.

Prima di comprendere il funzionamento della funzione "switch" è necessario definire le altre funzioni che parteciperanno alla conversione: convert_object, convert_array ed infine convert_other. Queste funzioni sono l'implementazione della funzione "converti" nello pseudocodice.

L'utilità delle suddette funzioni è descritta molto bene dai relativi nomi. Si mostra ora una figura, per capire in modo astratto i collegamenti tra le funzioni di cui si è parlato e il funzionamento dell'algoritmo.

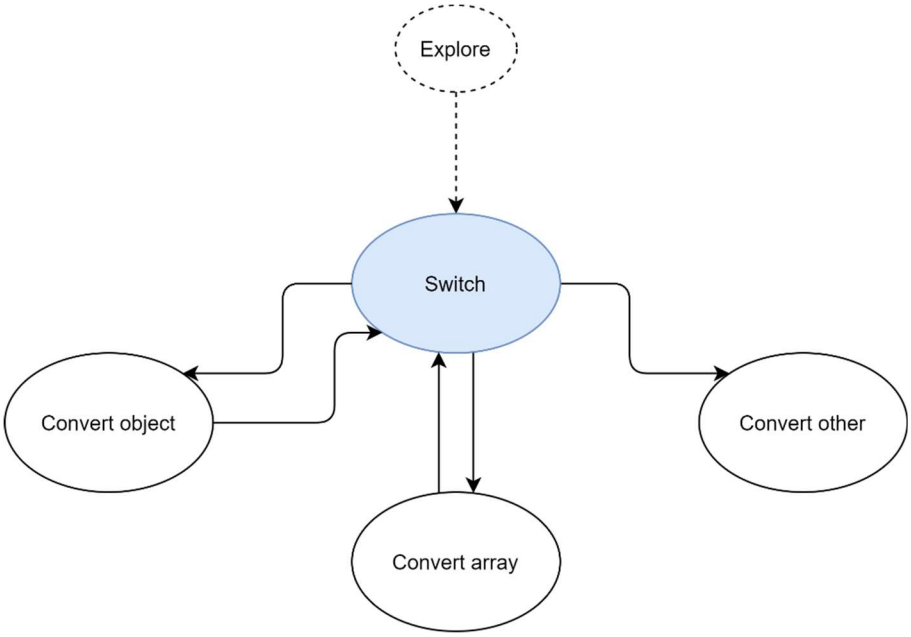

Figura 4.5.10: work-flow generazione dinamica del front end.

In generale, il funzionamento, come detto, inizia da "explore", che serve alla funzione "switch" un dizionario contenente lo schema dell'esame richiesto. La funzione "switch" cerca il primo "bsonType" disponibile e lo inoltra alla funzione corrispondente. Le funzioni sono così collegate: gli oggetti con "bsonType" "object" saranno ripartiti alla funzione "convert_object"; analogamente la funzione "convert_array" riceverà tutti gli oggetti il cui "bsonType" corrisponde ad "array"; tutti gli altri "bsonType", invece, vengono gestiti dalla funzione "convert_other".

Una volta che le funzioni di conversione ("convert_object" e "convert_array") ricevono il dizionario, lo analizzano e creano l'HTML corrispondente e nel caso trovino un sotto-dizionario, nonché un oggetto con proprio "bsonType", rimandano quel sotto-dizionario a "switch", che lo torna ad inoltrare alla funzione corrispondente. In questo modo le suddette funzioni continueranno a richiamarsi in modo ricorsivo, inoltrandosi porzioni sempre più piccole del dizionario iniziale (contenente lo schema completo), fino ad arrivare al punto in cui il dizionario sarà formato da un solo oggetto semplice, l'ultimo. Giunti all'ultimo elemento, la ricorsione terminerà. L'esecuzione percorrerà a ritroso l'albero generato dalla ricorsione, ed una volta arrivata in cima, riprenderà restituendo in risposta alla richiesta la pagina HTML appena generata.

Quando invece l'operazione richiesta è la visualizzazione, la generazione dell'HTML si compone di due fasi: l'analisi del documento contenente i dati e la generazione di

codice sopra descritta. L'operazione di visualizzazione richiede alcune operazioni in più, infatti, la creazione della pagina, in questo caso, necessita della fase di analisi del documento. In questo secondo caso di generazione dinamica del front end, la prima fase, ovvero l'analisi del documento, è direttamente collegata con la seconda, ovvero, la creazione dell'HTML sulla base dello schema dell'esame. Infatti, ogni volta che nella prima fase verrà analizzata un'informazione del documento, in base al titolo della stessa, verrà ricercata, mediante la seconda fase, lo schema di quel campo nella struttura totale dell'esame, ed una volta trovato, verrà generato l'HTML corrispondente e si passerà all'elemento successivo, fino alla fine del documento. Si può quindi affermare che parlare di due fasi in questo caso è errato, in quanto le due componenti (analisi e ricerca con conversione) lavorano quasi all'unisono per la generazione della pagina.

4.5.4 Suddivisione funzioni

In questa sezione verranno commentate e trattate le più importanti funzioni che compongono l'algoritmo per la generazione dinamica del front end. In particolare, essendo, le funzioni utilizzate per generare la pagina dell'inserimento dei dati, molto simili a quelle per la generazione della visualizzazione, verranno analizzate solo le prime: "switch", "convert_object", "convert_array" e "convert_other".

```
def switch(skeleton, bsonType, json_dict, key_name = '', array_mode = False, counter = 0):
    function = { 'object': convert_object, 'array': convert_array, 'string': convert_other, 'date': convert_other, 'number': convert_other}
    function[bsonType](skeleton, json_dict, key_name, bsonType, array_mode, counter = counter)
```

Figura 4.5.11: funzione switch.

Si riporta, in figura 4.5.11, il codice della funzione "switch". Si può notare come la variabile "function" contenga un dizionario le cui chiavi corrispondo ai "bsonType" più utilizzati, le quali corrispondono a loro volta al nome della funzione utile alla conversione del "bsonType" corrispondente. Il funzionamento della variabile appena descritta è molto simile al concetto di puntatore di funzioni, nel linguaggio di

programmazione C. Nella riga successiva, infine, viene effettivamente richiamata la funzione di conversione sulla base del “bsonType” estratto.

```
def convert_object(skeleton, sub_dict, key_name, bsonType, array_mode = False, counter = 0):
    skeleton.append('<fieldset>')
    if 'title' in sub_dict.keys():
        if counter > 0:
            skeleton.append('<label class="field_title">'+sub_dict['title'].replace("_"," ").capitalize()+'</label>')
    else:
        skeleton.append('<div class="div_flag">')
        if 'flag' in key_name.lower() and not key_name.lower() == 'flag':
            skeleton.append('<input type="checkbox" name="'+key_name+'$$checked" >')
        if not filter_name(key_name.replace("_","")).capitalize() == 'Flag':
            skeleton.append('<label class="field_title">'+filter_name(key_name.replace("_"," ")).capitalize()+'</label>')
        skeleton.append('</div>')
    counter += 1
    for p in sub_dict['properties'].items():
        switch(skeleton, p[1]['bsonType'], p[1], (key_name+'$$'+p[0]) if not key_name == '' else p[0], counter = counter)

    skeleton.append('</fieldset>')
```

Figura 4.5.12: funzione convert_object.

In figura 4.5.12, invece, è riportato il codice della funzione “convert_object”. In questa funzione si può vedere come lo scheletro della pagina HTML viene effettivamente creato. Il suddetto viene memorizzato come stringa in una lista Python, “skeleton”. Ogni elemento di tipo “object” viene incapsulato in un elemento HTML “fieldset” per una migliore impaginazione dei form di inserimento. Successivamente, dopo diversi controlli, mediante un ciclo for vengono iterati tutti gli oggetti contenuti nelle “properties” dell’elemento corrente, i quali, uno alla volta, vengono inoltrati, come detto prima, alla funzione “switch”.

La funzione “convert_array”, funziona in modo del tutto analogo a quella appena descritta, se non per un particolare: in questo caso il ciclo for itererà gli oggetti all’interno della chiave “items”.

Infine, la funzione “convert_other”, l’unica che non rimanda mai a “switch”, convertirà gli oggetti di tipo “string”, “number” e “date” in input HTML dello stesso tipo.

```
def convert_other(skeleton, sub_dict, key_name, bsonType, array_mode = False, counter = 0):
    if not key_name in ['patient_code','doc_code'] and not 'PE' in key_name:
        if bsonType == 'date':
            append_to_list(skeleton, 'date', key_name)
        elif bsonType == 'number':
            append_to_list(skeleton, 'number',key_name, minimum=(sub_dict['minimum'] if 'minimum' in sub_dict.keys() else 0), maximum=(sub_dict['maximum'] if 'maximum' in sub_dict.keys() else -1))
        elif bsonType == 'string':
            if 'file' in key_name:
                    append_to_list(skeleton, 'file', key_name)
            elif 'enum' in sub_dict.keys():
                skeleton.append('<div class="div_key_value"><label class="field_title">'+filter_name(key_name).replace("_"," ").capitalize()+'</label>')
                skeleton.append('<fieldset class="set_of_input">')

                if array_mode:
                    type_ = 'checkbox'
                else:
                    type_ = 'option'
                    skeleton.append('<select name="'+key_name+'">')

                #aggiungo tante checkbox o option quante possibili stringhe ho in enum
                for e in sub_dict['enum']:
                    append_to_list(skeleton, type_, key_name, value=e)

                if not array_mode:
                    skeleton.append('</select>')
                skeleton.append('</fieldset></div>')
            else:

                tmp = key_name.split('$$')
                if not tmp[-1] in ['annotazione','altro']:
                    if tmp[-1] == 'note' or 'note' in tmp[-1].lower():
                        skeleton.append('<div class="div_key_value" '+('style="display: none;"' if 'Flag' in key_name else '')+'>')
                        append_to_list(skeleton, 'textarea', key_name)
                        skeleton.append('</div>')
                    else:
                        append_to_list(skeleton, 'text', key_name)
                elif tmp[-1] == 'annotazione':
                    #caso annotazione
                    sub_name = key_name[0:key_name.index('annotazione')]
                    for s in range(len(skeleton)):
                        if 'name="'+sub_name in skeleton[s] and 'list' in skeleton[s]:
                            if 'name="'+sub_name+'list' in skeleton[s]:
                                skeleton[s] = skeleton[s].replace('name="'+sub_name,'onclick=enable_note(event) name="'+sub_name)
                elif tmp[-1] == 'altro':
                    name = tmp[0]
                    for t in tmp[1:len(tmp)-1]:
                        name += '$$'+t

                    for i in range(len(skeleton)-1,0,-1):
                        if 'name="'+name in skeleton[i] and "checkbox" in skeleton[i]:
                            skeleton[i] += ('<textarea name="'+key_name+'" placeholder="'+tmp[-1].capitalize()+'"></textarea>')
                            break
                        elif 'name="'+name in skeleton[i] and "select" in skeleton[i]:
                            for j in range(len(skeleton)-1,i,-1):
                                if '</select>' in skeleton[j]:
                                    skeleton[j] += ('<textarea name="'+key_name+'" placeholder="'+tmp[-1].capitalize()+'"></textarea>')
                                    break
                            break
```

Figura 4.5.13: funzione convert_other.

4.6 Implementazione download

In questa sezione si andrà a descrivere brevemente il back end delle operazioni di download. Per cominciare, all'interno del file "app.py", le funzioni che si occupano di questa operazione sono "download" e "select". La funzione "download", se richiamata senza argomenti (quindi dalla Home), restituirà la pagina mediante cui sarà possibile selezionare l'esame di cui scaricare gli esami. Una volta selezionato l'esame, la funzione "select" restituirà in risposta una pagina generata in modo analogo alla generazione delle pagine di inserimento e visualizzazione, per permettere all'utente di selezionare quali campi includere nel file CSV che verrà poi successivamente

scaricato. Una volta selezionati i suddetti campi, la funzione “download”, richiamata in questo caso passandogli come argomento il nome dell’esame selezionato ed i campi selezionati, si occuperà di creare il file CSV che verrà poi salvato sul PC dell’utente autenticato. Per generare il file CSV la funzione “download” recupera prima la lista di tutti i pazienti (linea 98, figura 4.6.1) e per ognuno cerca nella collection dell’esame selezionato un documento associato. Successivamente, dopo aver estratto dal file JSON lo schema corrispondente all’esame selezionato, verrà creato il file CSV, posizionando un campo della struttura per ogni colonna, a partire dalla colonna due. La prima colonna invece, verrà utilizzata per la lista dei pazienti, uno per ogni riga. L’ultima operazione è l’iterazione mediante for di tutti i documenti recuperati. Ad ognuno di questi corrisponderà un paziente della lista. Per ogni documento verranno quindi ricercati i campi posizionati nell’intestazione del file CSV, ed i dati trovati verranno posizionati, in corrispondenza dell’intestazione, nella riga corrispondente al paziente. Dal momento che tra le richieste era stata sottolineata l’importanza di avere nel file tutti i pazienti, anche quelli senza dati relativi all’esame selezionato, si è realizzata la funzione di creazione del CSV affinché tali pazienti risultassero nel file in corrispondenza di una riga vuota, se non nella prima colonna dove è salvato il codice del paziente stesso. Alla fine di questa sequenza di operazioni avremo creato un file rappresentabile come una tabella, le cui righe saranno i dati corrispondenti ai pazienti e le colonne i dati dell’esame selezionato.

```python
@app.route('/download/<path:path>', methods=['POST','GET'])
def download(path = None):
    if request.method == 'POST':
        dati_richiesti = request.form.getlist('download_list')

        riga = {}
        dizionario = {}

        # #lista di pazienti e dizionario chiave(codice paziente) valore(dati esame path del relativo codice->paziente)
        pazienti = mongo.db.patient.find({})
        for p in pazienti:
            code = p['patient_code']
            dizionario[code] = mongo.db[path].find_one({'patient_code':code})

        if path[-1].isdigit():
            path=path[:len(path)-1]
        with open('Json_file/'+path+'.json', 'r', encoding='utf-8') as f:     #apro il file json specificato in path
            jfile = json.load(f)
            jfile = jfile['$jsonSchema']['properties']

        aggiunta = []
        #per ogni chiave della lista
        for l in dati_richiesti:
            #salvo bsonType
            bsonType = []
            search_type_and_other_in_json(path, l.split('$$'), bsonType)

            #controllo ogni paziente
            for p in dizionario:
                if not p in riga:
                    riga[p] = {}
                #controllo tutte le componenti della chiave
                riga[p][l] = dizionario[p]

                for s in l.split('$$'):
                    #controllo che ci sia la chiave parziale
                    if riga[p][l] == None:
                        riga[p][l] = None
                        break
                    else:
                        if s in riga[p][l]:
                            riga[p][l] = riga[p][l][s]
                        else:
                            riga[p][l] = None
                convert_to_csv_list(riga,p,l,bsonType[-1],jfile, dati_richiesti, aggiunta)

        fieldnames = ['patient_code']+dati_richiesti+aggiunta
        with open('CSV_download/'+path.upper()+'_Data.csv', mode='w', newline='') as csv_file:
            writer = csv.DictWriter(csv_file, fieldnames=fieldnames)

            writer.writeheader()

            for p in riga:
                riga[p]['patient_code'] = p
                writer.writerow(riga[p])
        return send_file('CSV_download/'+path.upper()+'_Data.csv', as_attachment=True)
        # return send_from_directory(directory='CSV_download', filename=path.upper()+'_Data.csv')
    else:
        return render_template('download.html', msg_code=None)
```

Figura 4.6.1: funzione download.

Conclusioni

MCIBase oltre ad essere la prova di un'ottima collaborazione tra diversi dipartimenti di UNIMORE, è la realizzazione di un sistema in grado di adattarsi a diverse esigenze, nonché il raggiungimento dello scopo prefissato.

Il suddetto sistema infatti, oltre ad offrire un servizio solido quanto stabile per l'archiviazione e la manipolazione di dati neuroscientifici, è anche un applicazione web in grado di adattarsi a diverse situazioni non banali. Per com'è stata progettata ed implementata, nulla vieta in futuro, di poter modificare gli schemi e le strutture degli esami, infatti, il sistema dovrebbe essere in grado di assorbire le modifiche e mantenere la sua attuale integrità.

Per ciò che riguarda possibili sviluppi futuri inoltre, si è già pensato di inserire altre funzionalità, tra cui uno strumento per la ricerca avanzata, uno strumento per il caricamento di dati da file CSV ed una pagina per la visualizzazione di statistiche utili alla ricerca stessa. Ancora, grazie alla progettazione e ai componenti utilizzati per la realizzazione del progetto, l'applicazione si prospetta stabile anche nel caso di espansione della ricerca su territorio nazionale. Infine, per migliorare ulteriormente l'applicazione, sarebbe interessante integrare MCIBase con la Biobanca, dove verranno conservati tutti i campioni biologici dei pazienti reclutati.

MCIBase ai fini del progetto di ricerca risulterà utile per mantenere i dati un'unica locazione, affinché tutti i partecipanti possano accedervi in qualunque momento e da qualsiasi luogo. Inoltre, grazie all'interfaccia realizzata, permetterà all'equipe di tenere sott'occhio la quantità di esami raccolti, nonché di scaricarli per svolgervi sopra analisi esterne.

Considerando invece il progetto MCIBase da un punto di vista teorico e non principalmente legato all'ambito per cui è stato sviluppato, offre sicuramente un approccio differente ed efficace da quelli generalmente utilizzati. La sua logica e la sua struttura potrebbero infatti essere applicate a qualsiasi problema con caratteristiche variabili e mutabili nel tempo, permettendo agli sviluppatori stabilità nel tempo.

Bibliografia

- [1] Web Application: https://it.wikipedia.org/wiki/Applicazione_web
- [2] Architettura three-tier: https://it.wikipedia.org/wiki/Architettura_three-tier
- [3] Logica di business: https://it.wikipedia.org/wiki/Logica_di_business
- [4] Base di dati: https://it.wikipedia.org/wiki/Base_di_dati
- [5] NoSQL: https://en.wikipedia.org/wiki/NoSQL
- [6] Metadata: https://en.wikipedia.org/wiki/Metadata
- [7] Documentazione MongoDB : https://docs.mongodb.com/manual/
- [8] WSGI: https://it.wikipedia.org/wiki/Web_Server_Gateway_Interface
- [9] Jinja2: https://it.wikipedia.org/wiki/Jinja2
- [10] Licenza BSD: https://it.wikipedia.org/wiki/Berkeley_Software_Distribution

Ringraziamenti

Un *grazie sincero* è rivolto al Professore Riccardo Martoglia, Relatore di questa Tesi, per la disponibilità, la cortesia ed il rispetto dimostrati nei mei confronti.

Ancora, un *ringraziamento* va al mio Compagno di progetto Luca Sala, per la lealtà dimostrata durante lo sviluppo del progetto e della Tesi.

Infine, un *grazie speciale* va alla mia famiglia, ai miei parenti e ai miei amici che mi hanno sempre supportato ed incoraggiato durante tutto il percorso di studi.