UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Corso di Laurea in Informatica Musicale

Sistema di raccomandazione basato su Collaborative Filter per piattaforma Moon Cloud facente parte dell'ambito della Security Assurance

Relatore:
Prof. Claudio Agostino Ardagna
Correlatore:
Prof. Valerio Bellandi

Tesi di Laurea di:
Andrea Michele Albonico
Matricola: 886667

Anno Accademico 2018/2019

Ringraziamenti

Alla mia famiglia e a chi ha sempre creduto in me...

Andrea Michele Albonico

Prefazione

I sistemi di raccomandazione (*Recommendation System*) hanno avuto un forte sviluppo negli ultimi decenni e nascono proprio con lo scopo d'identificare quegli oggetti (detti generalmente *item*) all'interno di un vasto mondo d'informazioni che possono essere di nostro interesse e tanto maggiore è il grado di conoscenza dell'individuo e tanto più vengono ritenuti affidabili.
Il motivo di questo successo risiede nella riuscita integrazione di tali sistemi in applicazioni commerciali, soprattutto nel mondo dell'E-commerce e nel fatto che sono in grado di aiutare un utente a prendere una decisione, che sia la scelta di un film per l'uscita con gli amici il sabato sera, di una playlist da ascoltare durante un viaggio in auto o in un momento di lettura, e via discorrendo.
Moon Cloud è una piattaforma erogata come servizio che fornisce un meccanismo di *Security Governance* centralizzato. Garantisce il controllo della sicurezza informatica in modo semplice e intuitivo, attraverso attività di test e monitoraggio periodiche e programmate (*Security Assurance*). L'obiettivo di questa tesi è stato quello di aggiungere, al già presente sistema per la scelta dei Controlli all'interno delle attività di test, un sistema di raccomandazione che possa consigliare all'utente delle possibili *Evaluation* rispetto al target indicato; in questo modo anche l'utente meno esperto può usufruire dei servizi offerti da Moon Cloud in modo semplice e intuitivo.

La tesi è organizzata come segue:

Capitolo 1 – Introduzione a Moon Cloud in questo capitolo viene descritta la piattaforma Moon Cloud e il suo funzionamento in ambito di Security Assurance.

Capitolo 2 – Tecnologie utilizzate in questo capitolo vengono presentati gli studi e le analisi di soluzioni esistenti, studi delle tecnologie utilizzate per la realizzazione del progetto.

Capitolo 3 – Collaborative filtering in questo capitolo viene descritto in modo più approfondito gli studi compiuti sui Filtri Collaborativi

che hanno portato alla realizzazione dei sistemi di raccomandazione proposti nella soluzione implementata per la piattaforma Moon Cloud, inoltre verranno mostrate le relative porzioni di codice.

Capitolo 4 – Descrizione della soluzione in questo capitolo viene descritta in maniera dettagliata la realizzazione dell'applicativo, i risultati ottenuti e l'uso che se ne è fatto.

Capitolo 5 – Conclusioni in questo capitolo vengono esposte le conclusioni e i possibili sviluppi futuri delle attività svolte e del sistema realizzato.

Indice

Elenco delle figure

Capitolo 1

Scenario e motivazioni

In questo capitolo viene descritto in modo più approfondito il funzionamento della piattaforma Moon Cloud unitamente al motivo dell'implementazione della soluzione proposta.

1.1 Introduzione

La diffusione di sistemi *Information and Communications Technology* (definito anche con l'acronimo ICT) ha avuto luogo nella maggior parte degli ambienti lavorativi e privati in termini di servizi offerti, automazione di processi e incremento delle performance. L'uso di questa tecnologia ha assunto importanza a partire dagli anni novanta come effetto del boom d'Internet e al giorno d'oggi le professionalità legate almondo dell'ICT crescono in numero e si evolvono per specificità, per operare in ambienti fortemente eterogenei ma sempre più interconnessi fra di loro come il Cloud Computing, i Social Newtwork, il Marketing Digitale, i Sistemi IoT, la Realtà Virtuale, ecc.

In particolare, il Cloud Computing ha portato un rivoluzionario paradigma nella creazione di un nuovo business, virtuale e accessibile, in qualunque momento e luogo; esso sfrutta le tecnologie messe a disposizione dai sistemi ICT come le operazioni di virtualized computing, internet e distributed computing, provvedendo un sistema integrato molto potente. Google, Microsoft, Amazon sono un esempio di aziende che forniscono servizi di Cloud Computing in business ICT. Si può definire il Cloud Computing come l'abilità di accedere a risorse (come database o applicazioni) in poco tempo e in tutto il mondo attraverso una rete.
Gli immensi benefici del Cloud in termini di flessibilità, consumo delle risorse e gestione semplificata, lo rendono la prima scelta per utenti e industrie per

il deploy dei loro sistemi IT. Tuttavia il Cloud Computing solleva diverse problematiche legate alla mancanza di fiducia e trasparenza dove i clienti necessitano di avere delle garanzie sui servizi Cloud ai quali si affidano; spesso i fornitori di questi servizi non fornisco ai clienti le specifiche riguardanti le misure di sicurezza messe in atto.
Negli ultimi anni, sono state sviluppate tecniche e modi per rendere sicuri questi sistemi e proteggere i dati degli utenti, portando alla diffusione di approcci eterogenei che incrementarono la confusione negli utenti. Tecniche tradizionali di verifica della sicurezza basati su metodi di analisi statistica non sono più sufficienti e devono essere integrati con processi di raccolta di prove (in inglese *evidence*) da sistemi Cloud in produzione e funzionanti. In generale la *Cloud Security* definisce i modi, come criptazione e controllo degli accessi, per proteggere attivamente gli asset da minacce interne ed esterne, e fornire un ambiente in cui i clienti possano affidarsi e interagire in totale sicurezza.
Tutto questo non basta a rendere il Cloud fidato e trasparente, per questo sono state introdotte tecniche di *Security Assurance*, delle garanzie che permettono di ottenere la fiducia necessaria nelle infrastrutture e/o nelle applicazioni di dimostrare il rispetto di certe proprietà di sicurezza, e che operino normalmente anche se subiscono attacchi; grazie alla raccolta e allo studio di evidence è possibile che venga accertata la validità e l'efficienza delle proprietà di sicurezza messe in atto.
Il prezzo che si paga per i benefici di questa tecnologia è dato dall'incremento di violazioni di sicurezza, che oggigiorno preoccupa tutte le aziende e di conseguenza anche i loro clienti, con l'incremento del rischio di fallimento per i servizi più importanti dovuti a violazioni della privacy e al furto di dati. Il mercato sta lentamente notando che non è l'inadeguatezza tecnologica dei sistemi di sicurezza che incrementa il rischio delle violazioni; piuttosto, la mal configurazione e l'errata integrazione di questi sistemi nei processi di business [2].

1.2 Security Assurance

L'utilizzo di sistemi di sicurezza e di controllo migliori non garantisce in modo assoluto la sicurezza dell'infrastruttura; per garantire ciò è necessario implementare un processo continuo di diagnostica e verifica della corretta configurazione dei Controlli, supervisionando il loro comportamento, accertandosi che sia quello aspettato.

La *Security Assessment* diventa allora un aspetto importante specialmente negli ambienti Cloud e IoT. Questo processo, costituito da un insieme di attività mirate alla valutazione del rischio in sistemi IT, deve essere portato avanti in modo continuo e olistico, per correlare le evidence raccolte da sempre maggiori meccanismi di protezione [1].
In più, quando i sistemi Cloud e i servizi IoT sono coinvolti, le dinamiche di questi servizi e la loro rapida evoluzione rende il controllo dei processi all'interno dell'azienda e le politiche di sicurezza più complesse e prone ad errori.

I requisiti ad alto livello fondamentali per poter garantire la Security Assurance sono i seguenti.

Sistema olistico è richiesta una visione globale e pulita dello status dei sistemi di sicurezza; inoltre, è cruciale distribuire lo sforzo degli specialisti in sicurezza per migliorare il processo e le politiche messe in atto. Si parte da delle valutazioni fatte manualmente a quella semi-automatiche che vengono usate per ispezionare i meccanismi di sicurezza.

Monitoraggio continuo ed efficiente è necessario un controllo continuo che valuti l'efficienza dei sistemi di sicurezza per ridurre l'impatto dell'errore umano, soprattutto dal punto di vista organizzativo. La coesistenza di componenti in conflitto o la mancata configurazione dovuta al cambiamento dell'ambiente possono essere scenari che richiedono un monitoraggio e un aggiornamento continuo.

Singolo punto di management avere un solo punto d'accesso in cui poter gestire tutti gli aspetti relativi alla sicurezza, permette di avere sotto controllo le politiche di sicurezza. Inoltre, disporre di un inventario degli asset da proteggere permette di poter conoscere quali meccanismi di protezioni applicare.

Reazioni rapide a incidenti di sicurezza spesso la reazione a queste situazioni è ritardata da due fattori: il tempo richiesto per rilevare l'incidente e il tempo per analizzare il motivo dell'accaduto; e avere un sistema che implementa un monitoraggio continuo permette di venire a conoscenza di questi problemi in breve tempo e agire di conseguenza.

1.3 Moon Cloud

Moon Cloud è una soluzione PaaS (acronimo inglese di *Platform as a Service*) che fornice una piattaforma B2B (*Business To Business*) innovativa per verifiche, diagnostiche e monitoraggio dell'adeguatezza dei sistemi ICT, in modo continuo e su larga scala, rispetto alle politiche di sicurezza. Essa supporta una semplice ed efficiente *ICT Security Governance*, dove le politiche di sicurezza possono essere definite dalle compagnie stesse (a partire da un semplice controllo sulle vulnerabilità a linee guida di sicurezza interna), da entità esterne, imposte da standard oppure da regolamentazioni nazionali o internazionali. La sicurezza di un sistema o di un insieme di asset dipende solo parzialmente dalla forza dei singoli meccanismi di protezione isolati l'uno dall'altro; infatti, dipende anche dall'abilità di questi meccanismi di lavorare continuamente in sinergia per provvedere una protezione olistica.
Moon Cloud è un framework di *Security Assurance* il quale garantisce che un sistema ICT soddisfi certi requisiti prestabiliti da appropriate politiche e procedure precedentemente definite. Una *Security Compliance Evaluation* è un processo di verifica a cui un target è sottoposto e il cui risultato deve soddisfare i requisiti richiesti da standard e politiche. A partire da questi processi di controllo, che devono a loro volta essere affidabili, si ottengono delle evidence; queste ultime possono essere raccolte monitorando l'attività del target oppure, come già menzionato, sottoponendo il target a scenari critici o di testing.
In particolare, una Security Compliance Evaluation è un processo di verifica dell'uniformità di un certo target a una o più politiche attraverso una serie di Controlli che a seconda delle caratteristiche e proprietà del target, può avere successo o meno. Di conseguenza se un target supera tutti i Controlli a cui è sottoposto allora significa che rispetta la politica scelta. Moon Cloud imple-

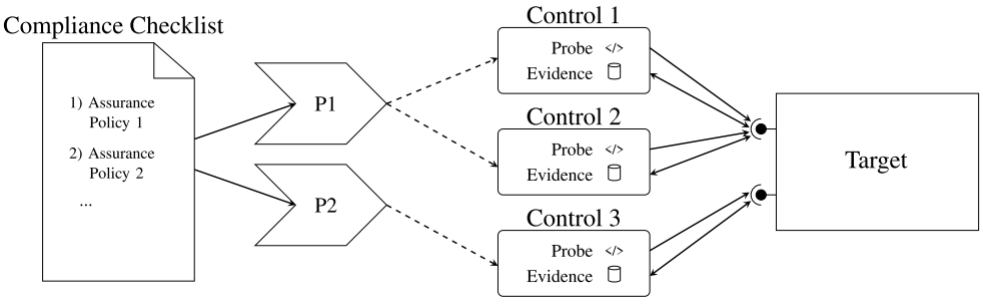

Figura 1.1: Security Compliance Evaluation

menta il processo di Security Compliance Evaluation in Figura 1.1 usando

Controlli di monitoraggio o di test personalizzabili. Inoltre, è dotato delle seguenti caratteristiche, le quali vanno a completare i requisiti elencati nella Sezione 1.2.

- Moon Cloud implementa un sistema di Security Assurance Evidence-based continuo, implementato come processo di Compliance, basato su politiche custom o standard; inoltre, presenta una visione olistica dello stato di sicurezza di un dato sistema.
- Moon Cloud permette di schedulare e configurare delle ispezioni automatiche, grazie all'inventario di asset protetto e senza l'intervento dell'uomo.
- Moon Cloud Evaluation Engine può ispezionare dall'interno un sistema, gestendo così delle minaccie interne; permettendo anche reazioni rapide a incidenti di sicurezza e veloci rimedi, grazie alla raccolta continua di evidence.

In generale, Moon Cloud gestisce i processi di Evaluation attraverso un set di *Execution Cluster*; ognuno dei quali gestisce ed esegue un set di *probe* che collezionano le evidence necessarie per effettuare i processi di valutazione. Tutte le attività di collezione sono eseguite dalla probe, ognuno dei quali è uno script Python fornito come una singola immagine di Docker, che viene inizializzata quando è triggerata una Evaluation ed è distrutta quando il processo di Evaluation è terminato.
Accedendo alla piattaforma di Moon Cloud, l'utente può definire le proprie politiche di sicurezza e attività di Evaluation come una serie di Controlli di sicurezza e altre politiche predefinite. Una volta che una politica viene definita, l'utente può decidere quando schedulare l'Evaluation; e nel momento in cui un processo di Evaluation viene inizializzato, tutti i Controlli e/o le politiche legate ad essa, vengono eseguiti e i risultati, raccolti dalla probe, vengono memorizzati e restituiti all'utente. A questo punto l'utente può accedere a questi risultati a diversi gradi di precisione: una visione sommaria e generale di tutte le politche implentate e dello stato generale del sistema di sicurezza, al risultato di una specifica politica oppure alle evidence raccolte per una Evaluation.

Per poter rendere ancora più intuitivo e semplice da utilizzare un sistema di questa importanza, si è pensato d'introdurre un sistema che possa raccomandare agli utenti, in base agli asset che vogliono proteggere e monitorare, una serie di Evaluation o politiche da applicare in quei casi; questo permette anche a utenti meno esperti di poter configurare in modo rapido ed efficiente meccanismi di protezione da minacce. Un sistema di raccomandazione permette

di selezionare all'interno di un ampio catalogo, un numero limitato di prodotti personalizzati sulla base delle preferenze dell'utente attivo. La ricerca in questo ambito si è sempre concentrata sulla qualità delle raccomandazioni di questi sistemi, tralasciando un aspetto fondamentale: la fiducia che un utente deve avere verso questi ultimi. E ciò è ottenibile se si è il più possibile trasparenti nei processi che portano alla nascita dei suggerimenti, partendo da questo si può ottenere l'ambita fiducia da parte degli utenti.

Capitolo 2

Tecnologie utilizzate

In questo capitolo sono descritte le tecnologie utilizzate per la realizzazione del progetto unitamente alle motivazioni legate all'uso di certi sistemi rispetto ad altri. In particolare viene fornita una panoramica dei sistemi di raccomandazione analizzati e studiati, nel capitolo successivo vengono approfonditi a livello pratico i sistemi di raccomandazione Memory-based i quali sono stati utilizzati per l'implementazione della soluzione.

2.1 Perché Python e Django

Python Python è un linguaggio di programmazione ad alto livello, orientato agli oggetti, adatto, tra gli altri usi, a sviluppare applicazioni distribuite, scripting, computazione numerica e system testing; ideato e rilasciato pubblicamente per la prima volta nel 1991 dal suo creatore Guido van Rossum, programmatore olandese.
Python supporta diversi paradigmi di programmazione, come quello object-oriented (con supporto all'ereditarietà multipla), quello imperativo e quello funzionale, e offre una tipizzazione dinamica forte. È fornito di una standard library estremamente ricca, che, unitamente alla gestione automatica della memoria e a robusti costrutti per il controllo delle eccezioni, rendono Python uno dei linguaggi più ricchi e comodi da usare; inoltre è anche semplice da usare e imparare. Python, nelle intenzioni di Guido van Rossum, è nato per essere un linguaggio immediatamente intuibile. La sua sintassi è pulita e snella così come i suoi costrutti, decisamente chiari e non ambigui.
Un aspetto inusuale e unico di Python è il metodo che viene usato per delimitare i blocchi di codice.

```
1 # Testing if two strings are equals
2 def test(got, expected):
3     if got == expected:
4         result = ' OK '
```

```
5       else:
6           result = ' X '
7       print('%s got: %s expected: %s' % (result, repr(got), repr(expected)))
8
9   def main():
10      test('hail', 'hailing')
11      test('swiming', 'swimingly')
12      test('do', 'do')
13
14  if __name__ == '__main__':
15      main()
```

Listing 2.1: Esempio di programma scritto in Python

Nei linguaggi come Pascal, C e Perl, i blocchi di codice sono indicati con le parentesi oppure con parole chiave (il C e il Perl usano ; il Pascal usa **begin** ed **end**). In questi linguaggi è solo una convenzione degli sviluppatori il fatto di indentare il codice interno ad un blocco, per metterlo in evidenza rispetto al codice circostante. In Python invece di usare parentesi o parole chiave, si usa l'indentazione stessa per indicare i blocchi nidificati in congiunzione col carattere "due punti" (:). Si usa sia una tabulazione sia un numero arbitrario di spazi, ma lo standard Python è di 4 spazi. Python è un linguaggio pseudo-compilato: un interprete si occupa di analizzare il codice sorgente (semplici file testuali con estensione .py) e, se sintatticamente corretto, di eseguirlo, non esiste una fase di compilazione separata (come, per esempio, avviene in C,) che generi un file eseguibile partendo dal sorgente [7].

Django Django è un web framework di alto livello basato su Pyhton che permette di sviluppare rapitamente e con tutti i presupposti per un sistema sicuro, un sito web perfettamente mantenibile. Esso si occupa della maggiori difficoltà dello sviluppo web, così da permettere di concentrarsi sulla scrittura dell'app; inoltre è open-source e ha una comunità attiva, una documentazione completa e semplice da consultare.
Django aiuta a scrivere applicazioni con le seguenti caratteristiche [3].

Versatile: può essere usato per la creazione di quasi tutti i tipi di siti web, a partire da sistemi per la gestione di contenuti, a social network e siti di notizie; può lavorare con qualunque client-side framework, e gestire contenuti in quasi tutti i formati (inclusi HTML, RSS feeds, JSON, Xml, etc). Internamente permette la scelta e l'implementazione di qualsiasi funzionalità (es. molti dei database più popolari, ecc.).

Sicuro: aiuta gli sviluppatori a evitare gli errori più comuni in merito alla sicurezza provvedendo un framework costruito per eseguire le operazioni in modo corretto e sicuro. Ad esempio, Django fornisce un metodo sicuro per gestire gli account degli utenti e le relative password, evitando errori comuni come inserire informazioni riguardanti la sessione

dell'utente nei cookies, dove sarebbero vulnerabili (invece i cookie contengono soltanto una chiave, e i valori effettivi sono salvati nel database) o salvare direttamente una password invece di una hash password.

Mantenibile: il codice di Django è scritto seguendo i principi e i pattern che incoraggiano la creazione di codice mantenibile e riusabile. Inoltre particolare, fa uso del principio "Don't Repeat Yourself" (DRY) così da ridurre al minimo le duplicazioni non necessarie, diminuendo la quantità di codice. Django raggruppa parti di codice letto in moduli seguendo le linee guida del pattern Model View Controller (MVC).

Portabile: Django essendo scritto in Python, un linguaggio multi piattaforma, lo rende indipendente dal sistema operativo eseguito sul server, che sia Linux, Windows o Mac OS X. Per di più, Django è ben supportato da molti web hosting provider, che spesso provvedono a specifche infrastrutture e documentazione per l'hosting di siti web in Django.

In generale, un tradizionale server web resta in attesa di richieste HTTP da parte del browser web (o altri client) degli utenti; nel momento in cui riceve una richiesta (solitamente di tipo POST o GET) l'applicazione legge le informazioni contenute in essa, all'interno dell'intestazione e/o del corpo, e nell'URL. A seconda della richiesta è possibile che vengano letti o scritti dati da un database o altre operazioni che portino al soddisfacimento della richiesta stessa, a questo punto l'applicazione restituisce una risposta al browser web dell'utente che ne ha fatto richiesta, spesso in modo dinamico, creando una pagina HTML, sulla base del Template, da mostrare, in cui può essere data la possibilità di inserire dati dall'utente in appositi campi compilabili o semplicemente mostrare delle informazioni.

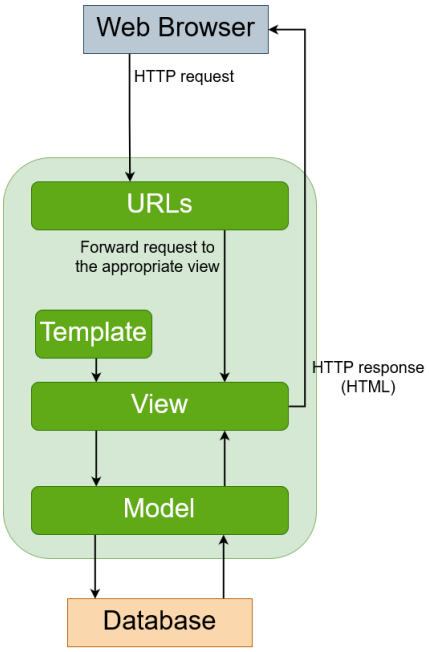

Figura 2.1: Schema generico di funzionamento di un applicativo web sviluppato con Django.

Un'applicazione web in Django tipicamente raggruppa il codice che gestisce questi passaggi in file separati, secondo la suddivisione in riquadri verdi della Figura 2.1 [5], in cui ognuno di questi gruppi di file svolge delle operazioni ben precise.

Un **URL mapper** è usato per reindirizzare le richieste HTTP alla View corretta in base all'URL della richiesta; è possibile processare richieste da qualsiasi URL attraverso una singola funzione, ma è più mantenibile scrivere diverse View per gestire ogni risorsa. Inoltre è possibile controllare se nell'URL è presente un particolare pattern di stringhe o numeri, e passare di conseguenza la richiesta alla funzione appropriata come dati da elaborare.

Una **View** è una funzione che gestisce le richieste HTTP, e restituisce una risposta HTTP. Le View accedono ai dati necessari per soddisfare la richiesta, anche attraverso i Model, e si delega la formattazione delle risposte ai Template.

I **Model** sono oggetti in Python che definiscono la struttura dei dati dell'applicazione, e provvedono meccanismi per gestirla (add, modify, delete) e query per interpellare il database.

Un **Template** è un file di testo che definisce la struttura o il layout di un altro file (come una pagina HTML), attraverso placeholder per rappresentare la posizione e il contenuto effettivo. Una View può creare dinamicamente una pagina HTML usando un Template HTML, popolandolo con dati presi dal Model, che a sua volta può recuperli dal database.

Nel Listing 2.1 si può osservare come viene scritto un URL e come si interfaccia con una View; in questo caso nel momento in cui viene fatta una richiesta HTTP a questo URL `recommendation/item/<str:item_other_id>/` viene richiamata la View `recommendation_views.item_recommendation` passando il parametro `<str:item_other_id>`.

```
# URL Example 'recommendation/item/35/'
path('recommendation/item/<str:item_other_id>/', recommendation_views.item_recommendation, name='
    item_recommendation')
```

A questo punto, come viene mostrato nel Listing 2.1, viene richiamata la View, essa prende i valori che gli vengono passati in ingresso e restituisce un risultato, in questo caso viene richiamata la View che implementa il sistema di raccomandazione Item-based, la quale si effettua una ricerca nel database per il valore del parametro `<str:item_other_id>` in ingresso, verificando l'esistenza di quell'item, questo è possibile grazie all'ausilio dei Model i quali sono stati utilizzati precedentemente per definire la struttura delle tabelle e dei relativi campi contenuti nel database e ora attraverso i cosiddetti QuerySet, messi a disposizione da Django, è possibile accedere a quei dati; successivamente vengono determinati tramite l'agoritmo di raccomandazione quali sono i possibili item simili e vengono salvati nella variabile `similar_item_evaluations`, infine, dopo aver ripulito i dati da informazioni poco rilevanti, viene restiuita una risposta in formato JSON al browser web che ha effettuato la richiesta.

```
# Item recommendation API REST
@api_view(['GET'])
def item_recommendation(request, item_other_id):
    # Trying to retrieve the actual node with item_other_id
    item = Evaluation.objects.get(other_id=item_other_id)

    similar_item_evaluations = item_recommendation_alg(item_other_id)

    # Cleaning the data, deleting all the keys except 'other_id'
    similar_item_evaluations_serilized = EvaluationSerializerRecommendation(similar_item_evaluations,
      many=True).data

    return JSONResponse(similar_item_evaluations_serilized, safe=False)
```

2.2 API REST

Quando si parla di REST (Representational State Transfer) si fa riferimento a un'architettura software, un termine introdotto per la prima volta nel

2000 all'interno di una tesi per il dottorato di Roy Fielding. Questo approccio architetturale venne ideato per creare web API basandosi sul protocollo HTTP. Il REST è infatti un sistema di trasmissione dei dati graie all'utilizzo del protocollo HTTP e dei suoi metodi: GET, POST, PUT, DELETE; grazie a quali si riesce ad identificare, accedere o modificare risorse ben precise.
Solitamente se si utilizzano delle API si svolgono delle operazioni all'interno di un database remoto:

- si effettuano richieste di tipo GET quando si vuole ottenere un determinato set di dati dal server.
- si effettuano richieste di tipo POST quando si vuole creare un nuovo oggetto all'interno del database.
- si effettuano richieste di tipo PUT quando si vuole modificare o sostituire completamente un oggetto già esistente.
- si effettuano richieste di tipo DELETE quando si vuole, da remoto, cancellare un oggetto contenuto all'interno del database al quale si è collegati.

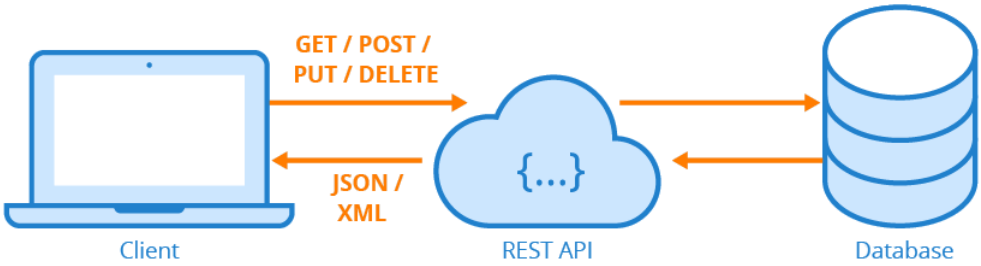

Figura 2.2: Schema generico di funzionamento di un'architettura REST.

Praticamente una buona API permette di fare tutte queste operazioni senza la necessità di dover accedere direttamente al database o alla soluzione implementata per l'inserimento, la modifica o il recupero delle informazioni.

2.3 Docker

Docker è una piattaforma software che permette di creare, testare e distribuire applicazioni con la massima rapidità. Docker raccoglie le applicazioni in unità standardizzate chiamate *Container* che offrono tutto il necessario

per la loro corretta esecuzione, incluse librerie, strumenti di sistema, codice e runtime. Con Docker, è possibile distribuire e ricalibrare le risorse per un'applicazione in qualsiasi ambiente, tenendo sempre sotto controllo il codice eseguito.
Questa tecnologia utilizza solitamente il kernel di Linux e le sue funzionalità, come Cgroups e namespace, per isolare i processi in modo da poterli eseguire in maniera indipendente. Questa indipendenza è l'obiettivo dei Container: la capacità di eseguire più processi e applicazioni in modo separato per sfruttare al meglio l'infrastruttura esistente pur conservando il livello di sicurezza che sarebbe garantito dalla presenza di sistemi separati.

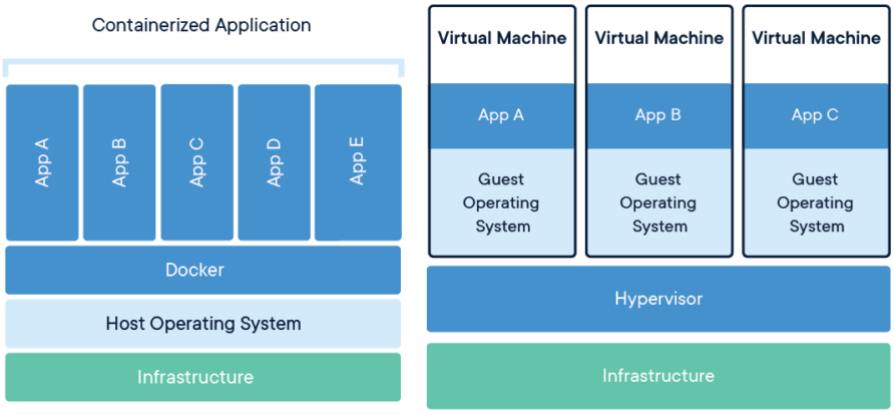

Figura 2.3: Schematizzazione di Container in Docker e di Virtual Machine.

Gli strumenti per la creazione di Container, come Docker, consentono il deployment a partire da un'*immagine*, ciò semplifica la condivisione di un'applicazione o di un insieme di servizi, con tutte le loro dipendenze, nei vari ambienti. Docker, considera i Container come macchine virtuali modulari estremamente leggere, offrendo la flessibilità di creare, distribuire, copiare e spostare i Container da un ambiente all'altro, ottimizzando così le app per il cloud.
I Container forniscono una modalità standard per impacchettare il codice delle applicazioni, le configurazioni e le dipendenze, in un singolo oggetto e condividono un sistema operativo installato sul server, operando come processi con risorse isolate, assicurando velocità, affidabilità e distribuzioni coerenti, indipendentemente dall'ambiente.
Questo sistema crea un livello di astrazione fra i Container e il sistema operativo ospitante e gestisce l'attivazione e la disattivazione dei Contenitori. Un'altra grande differenza è che la virtualizzazione permette di eseguire più sistemi operativi contemporaneamente in un singolo sistema, mentre i Con-

tainer condividono lo stesso kernel del sistema operativo e isolano i processi applicativi dal resto dell'infrastruttura.

2.4 Strutture dati gerarchiche

Nel caso di questa tesi si vogliono memorizzare due tassonomie, aventi struttura ad albero, in cui ogni nodo corrisponde, all'interno di una tabella, ad un record; quindi in dati gerarchici si instaurano delle relazioni padre-figlio tra le quali non possono essere rappresentate in modo naturale all'interno di un database relazionale, il quale, per l'appunto, segue il Modello Relazionale.
Esso è un modello logico di rappresentazione dei dati all'interno di un database, in cui ogni riga di una tabella è un record identificato univocamente da una chiave primaria, e le colonne contengono gli attributi dei dati e in genere ogni record ha un valore per ogni attributo.
In questa tesi si è lavorato su un database Sql, in cui i dati normalmente sono conservati come semplici "flat table", e in particolare si è usato il DBMS PostgreSQL, un sistema di gestione di database relazionali ad oggetti (ORDBMS). In generale le tabelle contenute in questo tipo di base di dati non permettono la memorizzazione secondo un modello gerarchico (come nell'Xml).
Per questo motivo è sorta la necessità di cercare un metodo alternativo per poter rappresentare queste strutture all'interno di database tradizionali. In questo caso ogni nodo ha un solo padre e nessuno o più figli (a eccezione del nodo radice che non ha un nodo padre); questo genere di rappresentazione delle informazioni, può essere trovato in diversi ambiti di applicazione di un database, incluse discussioni su forum e mailing list, grafici di organizzazione di un business, categorie per gestire contenuti e prodotti.
Durante lo studio compiuto per la realizzazione di questa testi sono stati analizzati diversi approcci per poter gestire le informazioni in modo gerarchico, i più importanti presi in considerazione sono i seguenti:

- The Adjacency List Model.
- The Nested Set Model.

2.4.1 The Adjacency List Model

Il primo approccio è chiamato *Adjacency List Model* o metodo ricorsivo; viene definito tale perché il suo funzionamento si basa su una funzione che itera per tutto l'albero.
In questo modello, ogni nodo dell'albero contenuto nella tabella ha associato

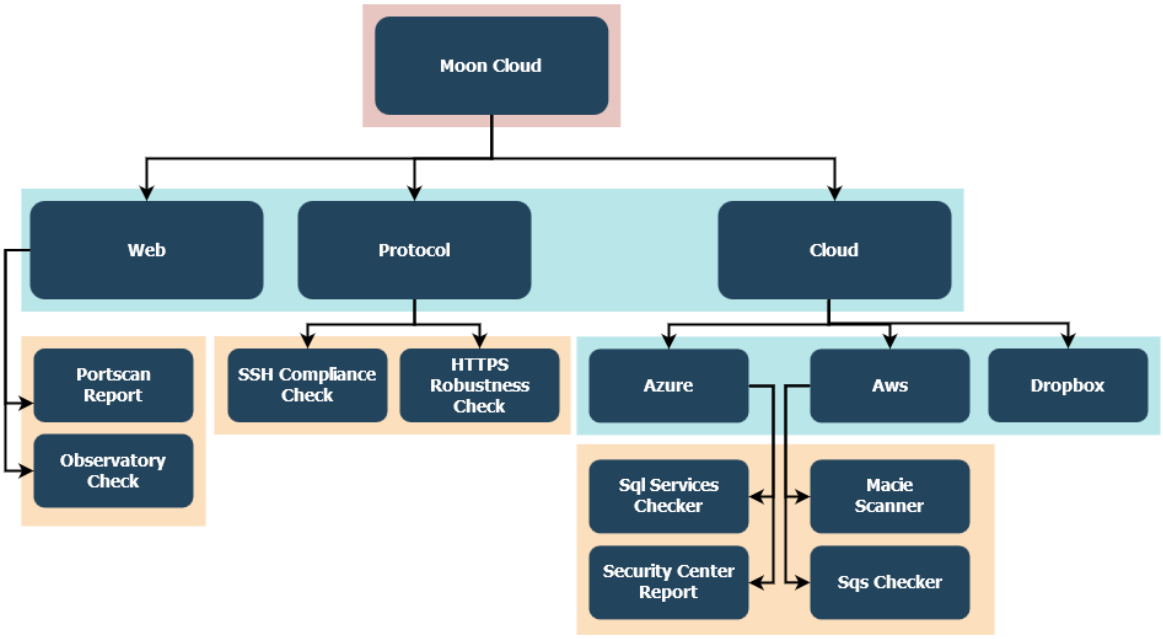

Figura 2.4: Esempio della rappresentazione gerarchica parziale dei dati nel progetto in questione.

un puntatore al suo nodo padre, e in particolare il nodo radice ha un puntatore a un valore NULL per quest'ultimo valore visto che è il nodo di partenza. La Tabella 2.1 mostra un esempio di possibile rappresentazione parziale dei dati nel database implementato in questo progetto secondo questo approccio, utilizzando come riferimento la Figura 2.4.
Il vantaggio di usare questo modello sta nella sua semplicità di costruzione, soprattutto a livello di codice client-side, e di restituzione dei figli di un nodo. Questo approccio diventa problematico nella maggior parte dei linguaggi di programmazione perché necessita di una query per ogni nodo dell'albero, e visto che ogni query impiega un certo periodo di tempo, questo rende la funzione molto lenta e poco efficiente quando si lavora con alberi di grandi dimensioni. Nel Esempio 2.2 è possibile osservare come viene recuparata in puro Sql l'intera tassonomia per le Evaluation; è possibile notare che la maggiore limitazione di questo approccio è che si necessità di un operazione di JOIN per ogni livello della gerarchia, e naturalmente questo porta a un degrado delle performance all'aumentare della complessità; nel caso di questo progetto si ha una tassonomia a tre livelli, quindi il problema è limitato, ma volendo avere una visione al futuro questo sistema col tempo diventerebbe sempre meno performante.

```
1 SELECT t1.name AS lev1, t2.name as lev2, t3.name as lev3
2     FROM Evaluation AS t1
```

id	name	parent
1	Moon Cloud	NULL
2	Web	1
3	Protocol	1
4	Cloud	1
5	Portscan Report	2
6	Observatory Check	2
7	SSH Compliance Check	3
8	HTTPS Robustness Check	3
9	Azure	4
10	Aws	4
11	Dropbox	4
12	Sql Services Checker	9
13	Security Center Report	9
14	Macie Scanner	10
15	Sqs Checker	10

Tabella 2.1: Esempio di una possibile tabella per gestire dati in modo gerarchico secondo l'Adjacency List Model.

```
3         LEFT JOIN Evaluation AS t2 ON t2.parent = t1.id
4         LEFT JOIN Evaluation AS t3 ON t3.parent = t2.id
5     WHERE t1.name = "moon cloud";
```

Listing 2.2: Query in puro Sql per recuperare l'intera tassonomia delle Evaluation, secondo l'Adjacency List Model.

Questa query si può tradurre in codice client-side attraverso una furnzione ricorsiva la quale determina per ogni nodo i suoi figli, per ogni figlio i suoi figli e così via finchè non si arriva ai nodi foglie. Inoltre, molti linguaggi non sono ottimizzati per funzioni ricorsive. Per ogni nodo, la funzione crea una nuova istanza di se stessa e ogni istanza occupa una porzione di memoria e impiega un certo tempo per inizializzarsi, più grande è l'albero e più questo processo sarà portato a termine in maggior tempo.

2.4.2 The Nested Set Model

Il secondo approccio analizzato è il *Nested Set Model*, il quale è stato utilizzato per l'implementazione delle tassonomie per le Evaluation e i Controlli (o politiche per verificare il soddisfacimento dei requisiti di sicurezza di un Target o asset indicato dall'utente) all'interno del progetto.
Questo approccio permette di osservare le gerarchie di dati in un modo diverso, non come nodi e linee, come se fosse un albero, ma come container innestati.

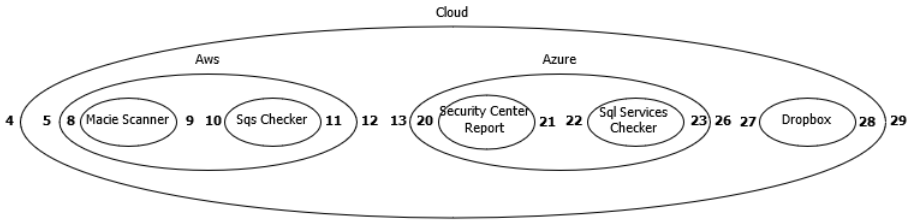

Figura 2.5: Esempio della gestione di dati in modo gerarchico secondo il Nested Set Model, utilizzando quelli presi dal database del progetto in questione (ridotto).

Con questo sistema la gerarchia viene mantenuta, secondo il principio cui un nodo padre contiene i suoi figli. Questa struttura viene mantenuta in tabella attraverso l'uso di due attributi aggiuntivi, `lft` e `rght`, come è possibile osservare dalla Tabella 2.2 seguente, la quale fa riferimento alla Figura 2.7 posta alla fine della sezione.

id	name	lft	rght
1	Moon Cloud	1	100
2	Web	86	99
3	Protocol	80	85
4	Cloud	4	29
5	Portscan Report	91	92
6	Observatory Check	89	90
7	SSH Compliance Check	83	84
8	HTTPS Robustness Check	81	82
9	Azure	13	26
10	Aws	5	12
11	Dropbox	27	28
12	Sql Services Checker	22	23
13	Security Center Report	20	21
14	Macie Scanner	8	9
15	Sqs Checker	10	11

Tabella 2.2: Esempio di una tabella per gestire dati in modo gerarchico secondo il Nested Set Model.

Dalla Tabella 2.2 la gerarchia dei dati viene rappresentata attraverso l'uso degli attributi *left* e *right* per rappresentare l'annidamento dei nodi (il nome delle colonne: *left* e *right*, hanno significati speciali in Sql; per questo motivo si identificano questi campi con i nomi `lft` e `rght`).

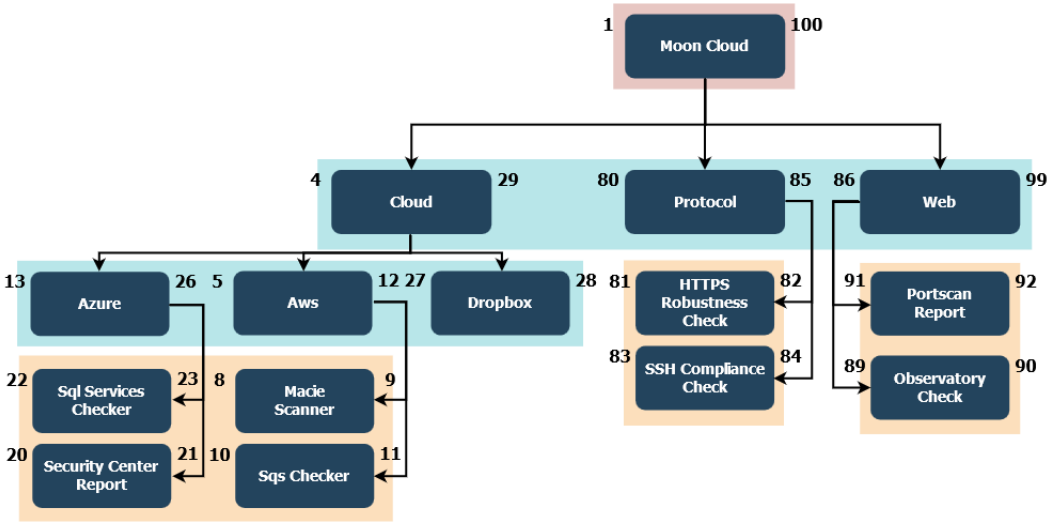

Figura 2.6: Esempio della gestione di dati in modo gerarchico secondo il Nested Set Model, utilizzando quelli presi dal database del progetto in questione.

L'assegnazione di questi valori viene effettuata ad ogni nodo visitandolo due volte e assegnando i valori in ordine di visita, e in entrambe le visite. Quindi vengono associati ad ogni nodo due numeri, memorizzati come due attributi. Più precisamente si inizia la visita dell'albero partendo da sinistra e continuando verso destra, un livello alla volta, scendendo per ogni nodo i suoi figli, assegnando i valori al campo *left*, prima di assegnare un valore al campo *right*, e successivamente si continua verso destra. Questo approccio è chiamato *Modified Preorder Tree Traversal Algorithm* (MPTT). A partire da questa tecnica è stato ideata la struttura della tassonomia delle evaluation e dei Controlli implementate nella soluzione proposta in questa tesi, con l'ausilio di un package di Python chiamato MPTT, del quale verrà illustrato il funzionamento nel capitolo 4.
Più semplicemente se si osserva la parte superiore della Figura 2.5 si può notare che la numerazione dei nodi, viene effettuata a partire da container più esterno da sinistra e continua verso destra.
A prima vista questo approccio può sembrare più complicato da comprendere rispetto all'Adjacency List Model, ma quest'ultimo è molto più veloce quando si vuole recuperare i nodi, visto che basta una query, mentre è più lento per operazioni di inserimento e cancellazione dei nodi; il Listing 2.3 qui di seguito mostra come è possibile recuperare l'itera tassonomia delle Evaluation; si può notare che questa query funziona in modo indipendente dalla profondità della tassonomia; inoltre, non è necessario preoccuparsi del valore

`rght` del nodo all'interno della clausola *BETWEEN* della query perché il valore cadrà sempre all'interno dello stesso nodo padre come anche il valore di `lft`.

```
SELECT node.name
    FROM Evaluation AS node, Evaluation AS parent
    WHERE node.lft BETWEEN parent.lft AND parent.rgt
        AND parent.name = "moon cloud"
    ORDER BY node.lft;
```

Listing 2.3: Query in puro Sql per recuperare l'intera tassonomia delle Evaluation, secondo il Nested Set Model.

Altro esempio è il caso in cui si vuole recuperare tutti i nodi foglia della tassonomia come mostra il Listing 2.4, in cui è ancora più semplice rispetto nell'Adjacency List Model. Nel Nested Set Model, il valori di `lft` e `rght` per i nodi foglia hanno valori consecutivi; quindi per trovare i nodi foglia basta cercare quei nodi dove il valore di `rght` è pari a quello di `lft` incrementato di uno.

```
SELECT name
    FROM Evaluation
    WHERE rgt = lft + 1;
```

Listing 2.4: Query in puro Sql per recuperare tutti i nodi foglia della tassonomia delle Evaluation, secondo il Nested Set Model.

Infine nel caso di inserimento o cancellazione di un nodo il grado di complessità dell'operazione è determinato dalla posizione del nodo che si vuole inserire o cancellare; a partire dal caso più semplice, quando si vuole inserire o cancellare un nodo foglia, quel nodo senza figli, fino al caso più complesso, quando si vuole cancellare un nodo padre perchè bisogna anche gestire i suoi nodi figli. Nel primo caso, è sufficente per poter cancellare un nodo senza figli eseguire una query come mostrata nel Listing 2.5, si determinano prima i valori dei campi `lft` e `rght` e la loro differenza, `width`, successivamente cancellato il nodo si sottrae la sua differenza da ogni nodo alla sua destra.

```
SELECT @myLeft := lft, @myRight := rgt, @myWidth := rgt - lft + 1
    FROM Evaluation
    WHERE name = "Sqs Checker";

DELETE FROM Evaluation WHERE lft BETWEEN @myLeft AND @myRight;

UPDATE Evaluation SET rgt = rgt - @myWidth WHERE rgt > @myRight;
UPDATE Evaluation SET lft = lft - @myWidth WHERE lft > @myRight;
```

Listing 2.5: Query in puro Sql per eliminare un nodo foglia dalla tassonomia delle Evaluation, secondo il Nested Set Model.

Nel secondo caso, in cui voglio eliminare il nodo padre ma non i suoi nodi figli si può decidere che i figli vengano spostati allo stesso livello del nodo padre eliminato, ciò viene mostrato dal Listing 2.6. In questo caso si sottrae due da tutti gli elementi a destra di tale nodo (visto che i figli avranno una dimensione, `width`, di due) e uno da tutti i nodi che sono suoi figli (per

chiudere il gap creato dalla perdita del nodo padre e dal valore associato al campo `lft`).

```
SELECT @myLeft := lft, @myRight := rgt, @myWidth := rgt - lft + 1
    FROM Evaluation
    WHERE name = "Web";

DELETE FROM Evaluation WHERE lft = @myLeft;

UPDATE Evaluation SET rgt = rgt - 1, lft = lft - 1 WHERE lft BETWEEN @myLeft AND @myRight;
UPDATE Evaluation SET rgt = rgt - 2 WHERE rgt > @myRight;
UPDATE Evaluation SET lft = lft - 2 WHERE lft > @myRight;
```

Listing 2.6: Query in puro Sql per eliminare un nodo padre dalla tassonomia delle Evaluation, secondo il Nested Set Model.

2.5 Sistemi di raccomandazione

Un sistema di raccomandazione (*Recommendation System*) è un sistema che consiglia a un utente uno o più item esistenti presenti in un database; l'*item* è inteso come una qualsiasi cosa di interesse all'utente, come prodotti, libri o giornali. Quando si esegue una raccomandazione si ha come obiettivo quello di consigliare l'item che possa essere tra quelli di maggiore interesse; in altre parole, devono essere in accordo con i gusti dell'utente.
Oggigiorno si possono trovare due principali trend di sistemi di raccomandazione.

Content-based filtering (CBF): un item viene raccomandato ad un utente se esso è simile ad altri item di interesse o piaciuti in passato, prendendo in considerazione prima gli item con alte valutazioni o quelli molto utilizzati; questo è possibile perché ad ogni item sono associate delle informazioni che lo descrivono, e questo insieme di dati viene definito metadati.

Collaborative filtering (CF): un item viene raccomandato ad un utente se i suoi vicini (altri utenti simili) sono interessati a quello stesso item.

Entrambi gli approcci hanno i loro punti di forza e di debolezza. Il primo algoritmo si focalizza sul contenuto degli item e sugli interessi del singolo utente e propone item differenti a utenti differenti, questo significa che ogni utente può ricevere raccomandazioni uniche. Tuttavia la più grande limitazione del CBF è il fatto di non poter determinare se un utente è interessato ad un item in modo implicito, perché analizza direttamente i metadati del prodotto e non considera gli interessi di altri utenti, i quali potrebbero suggerire item che non verrebbero notati con questo approccio. Per quanto riguarda il CF, nel caso siano presenti molti contenuti e proprietà associati agli item, allora vengono consumate molte risorse e tempo per poter analizzarli, nel contempo a questo algoritmo non interessano queste informazioni. Una raccomandazione viene fatta sulla base delle valutazioni degli utenti per gli item, o sugli usi che gli utenti fanno degli item e questo è il suo punto di forza perché non si trova a dover analizzare item ricchi di informazioni. Allo stesso tempo è anche il suo punto debole, perché può portare suggerimenti che potrebbero essere considerati poco adatti sulla base della poca relazione con i profili di alcuni utenti. Questo problema è accentuato quando sono presenti nel database molti item che non hanno valutazioni o non sono stati mai usati dagli utenti [4].

Un sistema di raccomandazione filtra i dati usando differenti algoritmi e rac-

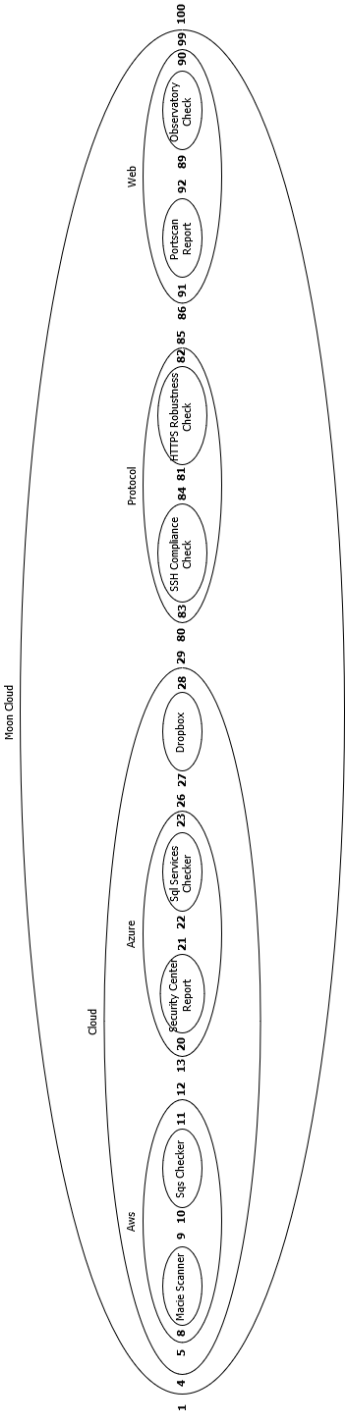

Figura 2.7: Esempio della gestione di dati in modo gerarchico secondo il Nested Set Model, utilizzando quelli presi dal database del progetto in questione.

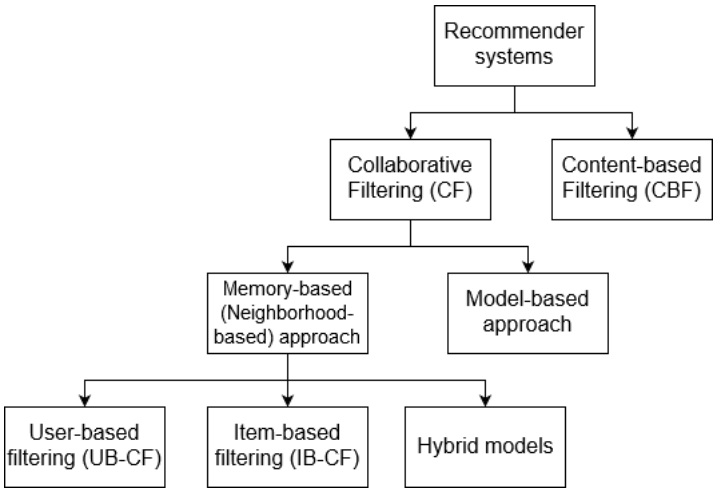

Figura 2.8: Categorizzazione generale dei sistemi di raccomandazione.

comanda gli item più rilevanti agli utenti attraverso un procedimento a 3 fasi.

Raccolta di dati: il primo step è anche quello più importante per poter costruire un sistema di raccomandazione che produca risultanti rilevanti e consistenti. I dati possono essere raccolti in due modi: esplicitamente, cioè attraverso la raccolta diretta di informazioni fornite dagli utenti, ad esempio le valutazioni di un prodotto oppure implicitamente. In questo caso vengono raccolti dati che, non sono prodotti in modo intenzionale dall'utente, ma sono ottenuti dai costanti flussi di dati come la cronologia di ricerca, i click effettuati, lo storico degli ordini, ecc.

Memorizzazione di dati: la quantità di dati definisce quanto efficace un modello di raccomandazione possa di diventare. Ad esempio, in un sistema di raccomandazione per film, maggiori sono le valutazioni fornite dagli utenti, e migliore sarà il sistema di raccomandazione per gli altri utenti. Il tipo di dati che si vuole raccogliere determina anche il supporto di memorizzazione più adatto.

Filtraggio dei dati: dopo la fase di raccolta e memorizzazione dei dati, essi vanno filtrati per poter estrarre le informazioni rilevanti e poter effettuare le raccomandazioni finali; inoltre, vi sono diversi algoritmi standard per realizzare quest'ultima fase.

2.5.1 Content-based filtering

Un Content-based filtering (definito anche con l'acronimo CBF) è un sistema di raccomandazione in cui vengono suggeriti, rispetto ad un item, quelli più simili, confrontando le informazioni contenute nei metadati, come una descrizione, uno o più autori, la categoria di appartenenza, ecc.. L'idea base che si trova dietro questi sistemi, è il fatto che se ad un utente piace o interessa un particolare item allora gli piaceranno anche altri con caratteristiche o proprietà simili.
Questo algoritmo suggerisce prodotti che piacevano all'utente nel passato ed è limitato a item dello stesso tipo. Un Content-based recommender fa riferimento a quegli approcci che provvedono raccomandazioni comparano la rappresentazione del contenuto che descrive un item e la rappresentazione del contenuto dell'item interessato dall'utente.
Questi metodi sono usati quando si conoscono a priori i metadati sugli item che si vuole suggerire, ma nulla sugli utenti. In questo sistema, delle *keyword* sono utilizzate per caratterizzare gli item e un profilo dell'utente è costruito per memorizzare quali item sono di suo interesse. In altre parole, questi algoritmi cercano di raccomandare quello che l'utente ha valutato positivamente o usato nel passato e sta esaminando nel presente. La costruzione del profilo dell'utente, spesso temporaneo, non viene basata su un modulo di registrazione che l'utente stesso deve compilare, ma su informazioni lasciate indirettamente dall'utente, le quali possono essere: i prodotti che ha maggiormente cercato e acquistato, quelli che sono stati inseriti nella lista dei desideri, ecc.. Più precisamente, tra vari item candidati da raccomandare all'utente si passa per un processo di confronto con gli item piaciuti dall'utente e gli item migliori vengono suggeriti.

2.5.2 Collaborative filtering

Il Filtraggio Collaborativo (definito anche con l'acronimo CF) per poter funzionare, si appoggia ad un database che raccoglie le preferenze degli utenti sulla base di un insieme di item, che a loro volta possono essere presenti nella stessa base di dati. Si sfruttano tecniche di analisi dei dati per poter ottenere delle raccomandazioni che consiglino gli utenti a trovare gli item che gli potrebbero piacere,eventualmente producendo una lista dei migliori N item. Un utente è sottoposto ad un processo di matching per poter scoprire quali sono i possibili *neighbours*, che corrispondono ai possibili utenti aventi storicamente delle preferenze in comune ad egli. Infine, gli item maggiormente preferiti dai *neighbours* sono raccomandati all'utente.
Questi sistemi tentano di predire la valutazione o la preferenza che un uten-

te darebbe a un item basandosi sulle preferenze date da altri utenti, queste ultime possono essere ottenute o in modo esplicito dagli utenti o tramite misurazioni implicite. Inoltre i Filtri Collaborativi non richiedono l'uso di metadati associati agli item, come nei Filtri Content-based.
Tuttavia, restano ancora oggi alcune sfide significative a cui sono sottoposti i sistemi di raccomandazione basati su Filtraggio Collaborativo.
Il primo obiettivo è quello di migliorare la scalabilità di questi algoritmi; essi sono in grado di cercare anche diecimila potenziali *neighbours* in tempo reale, ma la richiesta dei sistemi moderni è di cercare dieci milioni potenziali *neighbours*, per questo motivo possono nascere problemi di performance con i singoli utenti quando essi hanno molte informazioni associate.
Il secondo obiettivo è quello di migliorare la qualità dei sistemi di raccomandazione per gli utenti. Questi ultimi vogliono raccomandazioni di cui possono fidarsi e che possono aiutarli a trovare item di loro gusto e interesse.
Per certi versi questi due obbiettivi sono in conflitto tra di loro; per ottenere dei risultati validi e di una certa importanza è necessario trattarli in contemporanea perché aumentare solamente la scalabilità diminuirebbe la qualità delle raccomandazioni e viceversa [8].
Il principale modello di Filtri Collaborativi studiato in questo elaborato e approfondito nel capitolo successivo, è il metodo definito come *Memory-based* e il vantaggio di utilizzare questa tecnica sta nel fatto di essere semplici da implementare e i risultati ottenuti sono altrettanto semplici da interpretare; mentre si possono trovare anche Filtri Collaborativi che sfruttano metodi *Model-based* che si basano sulla fattorizzazione di matrici e sono molto più funzionali per gestire il problema della sparsità dei dati. Questi ultimi sono sviluppati usando algoritmi di data mining e machine learning per predire le valutazioni di utenti su item senza valutazioni, tentando di comprimere grandi database in un modello ed effettuare il processo di raccomandazione applicando dei meccanismi di riferimento all'interno di questo modello, questo permette ai CF Model-based di rispondere alle richieste degli utenti istantaneamente [4].

2.5.3 Il problema della Cold Start

Cosa succederebbe se un nuovo utente o un nuovo item venisse aggiunto al database? Questa situazione è chiamata *Cold Start* ed è possibile trovarne di due tipi.

Visitor Cold Start: si verifica quando un nuovo utente viene aggiunto al database, e visto che non è presente alcuno storico relativo, il sistema non è a conoscenza delle sue preferenze; per questo motivo diven-

ta molto più difficile raccomandare prodotti a quel particolare utente. Per risolvere questo problema, a livello teorico, si potrebbe applicare un procedimento di raccomandazione basata sulla popolarità dei prodotti, ma solo una volta che si è venuti a conoscenza delle preferenze dell'utente, sarà possibile generare delle raccomandazioni più precise e adeguate alle sue esigenze.

Item Cold Start: si verifica quando un nuovo item viene inserito nel sistema. L'azione dell'utente è quella più importante per determinare il valore di questo item all'interno dell'ecosistema; quindi maggiore è l'interazione che un item riceve maggiore è la facilità che venga raccomandato all'utente interessato.

Capitolo 3

Collaborative filtering

In questo capitolo vengono approfonditi gli algoritmi di raccomandazione implementati nella soluzione proposta, mostrando alcune porzioni di codice e approfondendo i vari passaggi che portano a ottenere delle raccomandazioni.

3.1 Memory-based

I Filtri Collaborativi Memory-based determinano per l'utente, che si sta considerando, quelle raccomandazioni che sono state fatte ad altri utenti che la pensano allo stesso modo. Questi metodi mirano a determinare il grado o il tipo di relazione tra utenti e item identificando o coppie d'item che tendono a essere usati insieme o che hanno un grado di similarità alto oppure utenti con uno storico di item usati simile [6]. Questi approcci divennero molto famosi grazie alla loro semplicità d'implementazione, inoltre sono molto intuitivi e non necessitano di operazioni di training sui dati e regolazione di molti parametri, permettendo all'utente di comprendere le ragioni che si celano dietro ad ogni raccomandazione.
Questa tipologia di sistemi di raccomandazione viene definita anche Filtri Collaborativi *Neighborhood-based* ed è possibile ulteriormente suddividerla in tre sottocategorie: User-based Filter, Item-based Filter e Hybrid Filter.

3.1.1 User-based filtering

Il sistema Filtraggio Collaborativo User-based, definito anche con l'acronimo UB-CF (*User-based Collaborative Filter*), basa tutto il suo funzionamento sulla comunità di utenti, maggiore è la sua dimensione e l'attività degli utenti con item o servizi e migliori possono essere le raccomandazioni. Questo algoritmo fornisce dei suggerimenti a un utente sulla base di uno o più vicini

(*neighbours*), e la similarità tra utenti può essere determinata sulla base degli item che l'utente ha utilizzato o valutato.
Molti di questi approcci possono essere generalizzati dall'algoritmo organizzato nei seguenti passi.

1. Specificare quale sia l'utente a cui si vuole applicare l'algoritmo di raccomandazione e recuperare i relativi utenti che possono avere dato valutazioni o usato item simili al primo utente. Per velocizzare l'esecuzione dell'algoritmo, piuttosto che recuperare tutti gli utenti, è possibile selezionare soltanto un gruppo di utenti in modo casuale oppure associare dei valori di similarità tra tutti gli utenti e confrontando questi valori con quello dell'utente target, selezionare i relativi utenti che superano una soglia scelta, oppure utilizzare tecniche di clustering.

2. Estrarre gli item con cui il primo utente non ha mai interagito e per questo motivo gli possono interessare, e mostrarli sotto forma di raccomandazioni.

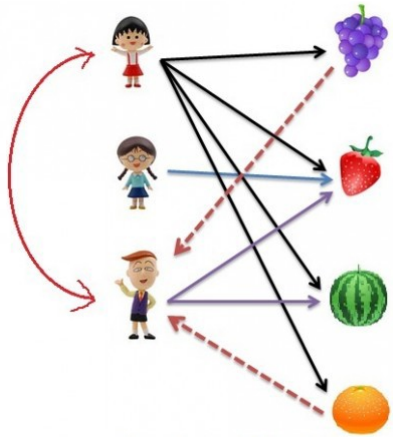

Figura 3.1: Esempio di applicazione di un sistema di raccomandazione User-based.

Questi approcci sono indipendenti dal contesto in cui sono applicati e possono essere più accurati rispetto a delle tecniche basate sul Content-based Filtering; dall'altra parte all'aumentare del numero di utenti che si considerano per effettuare le raccomandazioni migliore è la precisione di questo processo ma maggiore è il costo in termini di tempo.
Nella soluzione proposta in questa tesi, l'algoritmo UB-CF viene implementato sotto forma di funzione che prende in ingresso un parametro `user_other_id`,

come è possibile osservare dal Codice 3.1, corrispondente all'identificativo dell'utente, e restituisce una lista di raccomandazioni `similar_user_evaluations` corrispondenti alle Evaluation simili a quelle usate da altri utenti.
Per Evaluation si intende quel processo di verifica di uniformità di un certo target o asset, fornito dall'utente, a una o più politiche attraverso una serie di Controlli che a seconda delle caratteristiche e proprietà del target, può avere successo o meno. In altre parole, si può dire che un Evaluation è costituita da uno o più Controlli.

```
# User recommendation algortihm
def user_recommendation_alg(user_other_id):
```

Listing 3.1:

Più precisamente il funzionamento dell'algoritmo si svolge come segue.

- il primo passo è quello di recuperare, sulla base del parametro in ingresso alla funzione *user_other_id*, tutte le Evaluation utilizzate dall'utente in questione;

```
        # Select the target user and its evaluations
        target_user_evaluations = User.objects.get(other_id=user_other_id).evaluations.all()\
                                                                          .values('other_id', 'parent_id')\
                                                                          .order_by('other_id')

```

- il secondo passo consiste nel selezionare le Evaluation usate dagli altri utenti, e creare una lista di queste Evaluation (*other_users_evaluations*);

```
        # Select all other users and theirs evaluations
        other_users = User.objects.exclude(other_id=user_other_id)
        # Creating a list with all the evaluations of other users
        other_users_evaluations = []
        for o_users_evaluation in other_users:
            for evaluation in o_users_evaluation.evaluations.all().values('other_id', 'parent_id').order_by('other_id'):
                other_users_evaluations.append(evaluation)

```

- il terzo passo consiste nell'andare a determinare quali tra le Evaluation, dell'utente a cui si vuole raccomandare, quali sono quelle simili usate dagli altri utenti. Per determinare le Evaluation simili si è andato a confrontare il parametro (`parent_id`, associato ad ogni Evaluation), che identifica all'interno della base di dati quale sia il nodo padre per quella Evaluation, con lo stesso parametro delle restanti Evaluation; in questo modo si è andati a selezionare soltanto gli item appartenenti a una stessa categoria, e durante questo processo vengono eliminanti eventuali nodi duplicati. Infine viene composta una lista finale `similar_user_evaluations` con le Evaluation restanti. In definitiva ciò che ritorna questa funzione sono due liste: `target_user_evaluations`, che contiene le Evaluation usate dall'utente in questione e `similar_user_evaluations`.

```
        # Comparing target user's evaluations and other user's evaluations, and if there is a match the evaluation is
        # added to the 'similar_evaluations' list (the matching is made comparing the 'parent_id')
        similar_user_evaluations = []
        for t_user_evaluation in target_user_evaluations:
            for o_users_evaluation in other_users_evaluations:
                # Taking only the evaluations that have: different other_id (excluding the target evaluation
                # in the recommendation) and same parent_id and the evaluations that weren't added to 'target_user_evaluations'
                # list and to 'similar_user_evaluations'
                if ((t_user_evaluation['other_id'] != o_users_evaluation['other_id']) and  # Evaluations must have different 'other_id'
                        (t_user_evaluation['parent_id'] == o_users_evaluation['parent_id']) and  # Evaluations must have the same 'parent_id'
                        # Evaluation in all_other_evals list mustn't be already added to \
                        not (o_users_evaluation in target_user_evaluations) and  # the 'target_user_evaluations' list or
                        not (o_users_evaluation in similar_user_evaluations)):  # the 'similar_user_evaluations' list
                    similar_user_evaluations.append(o_users_evaluation)

        return target_user_evaluations, similar_user_evaluations

```

Nel capitolo successivo vengono mostrati degli esempi pratici in cui è stato applicato questo algoritmo.

3.1.2 Item-based filtering

Quando l'algoritmo UB-CF viene applicato per milioni di utenti e item non è molto efficiente per via della complessa computazione della ricerca di utenti simili. Per questo motivo venne ideata come alternativa il sistema Filtraggio Collaborativo Item-based, definito anche con l'acronimo IB-CF (*Item-based Collaborative Filter*) dove si è preferito evitare di confrontare tra utenti simili, e al suo posto viene effettuato un confronto tra gli item dell'utente a cui si vuole raccomandare e i possibili item simili.

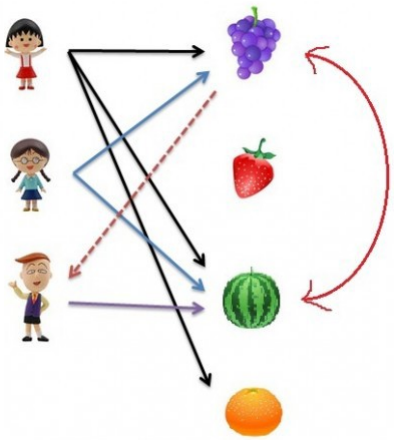

Figura 3.2: Esempio di applicazione di un sistema di raccomandazione IB-CF.

Questi sistemi sono estremamente simili ai sistemi di raccomandazione Content-based, e identificano item simili in base a come sono stati usati dagli utenti in passato [8].
A livello pratico nella soluzione proposta, questo algoritmo è stato implementato come funzione che ha un parametro `item_other_id`, in ingresso, rappresentante l'`other_id`, un attributo associato ad ogni item all'interno della base di dati che lo identifica, del item su cui si vuole andare a cercare altri item simili. In generale per determinare la similarità tra due oggetti si osserva l'attributo `parent_id` associato a ogni item, che determina quale sia il nodo padre tra tutti i nodi all'interno del database, in sostanza si selezionano quegli item che appartengono alla stessa categoria.
In generale il IB-CF ideato per determinare Evaluation simili funziona seguendo i seguenti passi.

- il primo passo è quello di recuperare sulla base del parametro in ingresso alla funzione `item_other_id` l'Evaluation su cui si vuole determinare le altre Evaluation simili;

```
    def item_recommendation_alg(item_other_id):
        # Selecting the evaluation, which is applied this algorithm, from its other_id
        # SELECT * FROM recommendation_app_evaluation WHERE other_id = %(item_other_id)s AND node_type = 'eva'
        target_eval = Evaluation.objects.filter(Q(other_id=item_other_id) & Q(node_type="eva"))\
                                        .values('other_id', 'parent_id')[0]

```

- il secondo passo è quello di recuperare tutte le Evaluation, escludendo la prima recuperata, presenti nella base di dati;

```
        # Selecting the other evaluations, excluding the target evaluation
        # SELECT * FROM recommendation_app_evaluation WHERE other_id != %(item_other_id)s AND node_type = 'eva'
        all_other_evals = Evaluation.objects.filter(~Q(other_id=item_other_id) & Q(node_type="eva"))\
                                        .values('other_id', 'parent_id').order_by('other_id')

```

- il terzo e ultimo passo consiste nel andare a determinare le Evaluation che hanno lo stesso `parent_id`, quindi quelle appartenenti alla stessa categoria, dell'Evaluation ottenuta nel primo passo; inoltre, se presenti, vengono eliminati eventuali duplicati; e la funzione ritorna una lista `similar_item_evaluations` contente le Evaluation simili.

```
        # Creating a list with all the evaluations that are similar to the target evaluation (comparing the parent_id)
        similar_item_evaluations = []
        for evaluation in all_other_evals:
            # Taking only the evaluations that have: different other_id (excluding the target evaluation
            # in the recommendation) and same parent_id and the evaluations that weren't added to similar_item_evaluations
            # list
            if ((target_eval['other_id'] != evaluation['other_id']) and  # Evaluations must have different 'other_id'
                    (target_eval['parent_id'] == evaluation['parent_id']) and  # Evaluations must have same 'parent_id'
                    # Evaluation in all_other_evals list mustn't be already added to \
                    not (evaluation in similar_item_evaluations)): # the ' similar_item_evaluations' list
                similar_item_evaluations.append(evaluation)

    return similar_item_evaluations

```

Altro algoritmo del tipo IB-CF implementato in questa tesi sulla falsa riga di quello appena riportato nel Listing 3.1.2, è quello ideato per determinare quali Evaluation possono essere raccomandate per un Target inserito da un utente tra quelli supportati da Moon Cloud (Host avente Windows come sistema operativo, Host avente Linux come sistema operativo, sistemi che sfruttano servizi di Aws o di Azure e URL di siti web).
In Python questo algoritmo viene implementato come funzione che prende in ingresso l'identificativo univoco (`id`) del Target, e restituisce l'insieme delle Evaluation raccomandate per quel Target. Il funzionamento dell'algoritmo si svolge come segue:

- il primo passo è quello di recuperare tutte le Evaluation presenti nel database;

```
    def target_recommendation_alg(target_id):
        # Retriving all the evaluations in the database
        evaluations = Evaluation.objects.filter(node_type="eva")

```

- il secondo e ultimo passo è quello di andare a determinare quali sono i Controlli che hanno il valore dell'attributo `target_type_id` pari al parametro in ingresso della funzione `target_id`, e da quei Controlli determinare le Evaluation che li utilizzano, eliminando eventuali duplicati; determinando così le possibili Evaluation applicabili per quel Target.

```
        # Saving in the target_evaluations list the evaluations which controls have target_type_id equal to target_id
        target_evaluations = []
        for evaluation in evaluations: # Scanning all the evaluations
            for evaluation_controls in evaluation.controls.filter(target_type_id=target_id):
                if not(evaluation in target_evaluations): # Excluding evaluations duplicated
                    target_evaluations.append(evaluation)

        # Converting the Evaluation model's instance in a dict and putting the evaluation, as a dict, in a list
        target_evaluations_serializer = EvaluationSerializer(target_evaluations, many=True)

        return target_evaluations_serializer.data

```

Nel capitolo successivo vengono mostrati anche degli esempi dei valori di risposta di queste funzioni.

3.1.3 Hybrid Filtering

Nei Sistemi di Raccomandazione Ibridi si tende a voler combinare più tecniche di raccomandazione, raggruppando i pregi di ciascun approccio; infatti se si comparano i Sistemi di Raccomandazione Ibridi con quelli Collaborativi o Content-based, la precisione dei suggerimenti è solitamente maggiore.
Nella soluzione proposta in questa tesi, questo algoritmo viene direttamente implementato come API REST, alla quale vengono passati come parametri la `request`, l'oggetto HTTP che il browser invia al server contenente la richiesta HTTP (attraverso un particolare URL) e lo `user_other_id`, un valore recuperato come parametro dall'URL e rappresenta l'`other_id`, un attributo associato a ogni utente che rappresenta un identificativo per l'utente stesso. Inoltre il tutto viene limitato a essere richiamato solo tramite richieste HTTP con metodo GET.
Nel Codice 3.2 si possono vedere come vengono limitate le richieste al metodo GET e come viene definita la funzione.

```
@api_view(['GET'])
def hybrid_recommendation(request, user_other_id)
```

Listing 3.2:

Il funzionamento di questo algoritmo si svolge nei seguenti passi.

- il primo passo è quello di verificare se l'utente esiste nel database altrimenti viene generata un'eccezione (o errore) che è gestita in modo personalizzato, generando una risposta HTTP con codice di errore 404 (Not Found);

```
        # Trying to retrive the actual User with user_other_id
        user = User.objects.get(other_id=user_other_id)

```

- il secondo passo è applicare l'algoritmo di User Recommandation, descritto nella sezione precendete, per le Evaluation e ottenere due liste, la prima (`target_user_evaluations`) contenente le Evaluation che l'utente ha utilizzato, mentre nella seconda (`similar_user_evaluations`) si hanno le Evaluation che gli altri utenti utilizzano e simili alle Evaluation del primo utente;

```
        # Taking from the user_recommendation_alg the evaluation recommended from this approach (similar_user_evaluations)
        # and the user's evaluations (target_user_evaluations)
        target_user_evaluations, similar_user_evaluations = user_recommendation_alg(user_other_id)

```

- il terzo passo consiste nell'applicazione dell'algoritmo Item-based per ogni Evaluation usata dall'utente in questione così da ottenere delle raccomandazioni che sono compatibili con le Evaluation usate dall'utente; la similarità o appartenenza alla stessa categoria viene ottenuta osservando il valore del `parent_id`; anche in questo caso vengono eliminati eventuali duplicati e viene formata una lista (`similar_item_evaluations`) contenente le Evaluation simili ottenute dall'applicazione dell'algoritmo di raccomandazione Item-based;

```
        # For every evaluation used by users is extracted all other possible evaluations that have the same 'parent_id'
        similar_item_evaluations = []
        for t_user_evaluation in target_user_evaluations:  # for every target user's evaluations
            for item_evaluation in item_recommendation_alg(t_user_evaluation['other_id']):  # is applied the item_recommendation algorithm
                # Taking only the evaluations that have: different other_id (excluding the target evaluation
                # in the recommendation) and same parent_id and the evaluations that weren't added to 'similar_item_evaluations'
                # list or to 'similar_user_evaluations' or to 'target_user_evaluations'
                if ((t_user_evaluation['other_id'] != item_evaluation['other_id']) and # Evaluations must have different 'id'
                        (t_user_evaluation['parent_id'] == item_evaluation['parent_id']) and # Evaluations must have the same 'parent_id'
                        # Evaluation in all_other_evals list mustn't be already added to \
                        not (item_evaluation in similar_item_evaluations) and # the 'similar_item_evaluations' list,
                        not (item_evaluation in similar_user_evaluations) and # the 'similar_user_evaluations' list or
                        not (item_evaluation in target_user_evaluations)): # the 'target_user_evaluations' list
                    similar_item_evaluations.append(item_evaluation)

```

- il quarto passo consiste nel raggruppare le due liste contenenti le Evaluation raccomandate per l'utente secondo l'applicazione dei due algo-

ritmi, eliminando anche eventuali duplicati, così da ottenere un'unica lista (`similar_evaluations`) la quale viene ritornata dalla funzione sotto forma di risposta HTTP in formato JSON;

```
1         # Putting together the evaluations recommended in similar_user_evaluations list and
      similar_item_evaluations list
2         similar_evaluations = []
3         # Adding to similar_evaluations list the evaluation in the similar_user_evaluations
      list
4         for s_user_evaluation in similar_user_evaluations:
5             similar_evaluations.append(s_user_evaluation)
6         # Adding to similar_evaluations list the evaluation in the similar_item_evaluations
      list
7         for item_evaluation in similar_item_evaluations:
8             # Taking only the evaluations that weren't added to \
9             if (not (item_evaluation in similar_evaluations) and  # the 'similar_evaluations'
      list or
10                    not (item_evaluation in target_user_evaluations)):  # the '
      target_user_evaluations' list
11                similar_evaluations.append(item_evaluation)
12        similar_evaluations = sorted(similar_evaluations, key=lambda i: i['other_id'])
13
14        return JSONResponse(similar_evaluations, safe=False)
15
```

Nel capitolo successivo viene mostrato un esempio di risposta per quando si effettua una chiamata a questa funzione, ed è approfondito il contesto che è stato costruito attorno agli algoritmi di raccomandazione descritti in questo capitolo.

Capitolo 4

Descrizione della soluzione

In questo capitolo è approfondito l'aspetto puramente pratico e le fasi che hanno portato alla realizzazione della soluzione; inoltre vengono mostrate le applicazioni pratiche degli aspetti teorici enunciati nei capitoli precedenti.

La soluzione proposta in questa testi implementa un servizio di API REST, che si appoggia a un database Postgres, accessibile attraverso apposite URL; questo servizio permette di effettuare richieste al sistema di raccomandazione e di effettuare degli aggiornamenti a questa base di dati.

Preparazione della base di dati

Prima di poter costruire il sistema di raccomandazione proposto in questa tesi, sono state eseguite delle operazioni preliminari per poter impostare il progetto di Django e la relativa applicazione che implementerà effettivamente la soluzione. Come descritto nei capitoli precedenti per procedere alla costruzione di un sistema di raccomandazione bisogna avere a disposizione una base di dati solida da cui attingere tutte le informazioni; ed è proprio questo il primo passo che è stato seguito, disegnare e progettare un database da cui partire per la realizzazione degli algoritmi proposti. In generale Moon Cloud possiede una struttura delle Evaluation e dei Control ad albero, di conseguenza anche le tabelle del database rispecchiano questa struttura, partendo dalle considerazioni fatte sulle tecniche descritte nei capitoli precedenti si è deciso, che per implementare un database relazionale che gestisse dati gerarchici, di utilizzare la tecnica definita come *Modified Preorder Tree Traversal Algorithm*, la quale permette di massimizzare l'efficienza nelle operazioni di

recupero dei dati, e quindi velocizzare i processi di raccomandazione, ma scendendo a compromessi per quanto riguarda le operazioni d'inserimento e spostamento dei nodi all'interno della struttura. Il package MPTT è una app di Django che ha come obiettivo quello di semplificare il più possibile la realizzazione dei Model, utilizzati per la generazione della base di dati, e la gestione della struttura di dati ad albero; si prende cura di tutti i dettagli riguardanti la realizzazione delle tabelle e dei campi *left* e *right* associati ad ogni nodo della tassonomia, mettendo a disposizione dei tool per poter lavorare con le istanze dei Model. Nel Listing 4.1 è possibile trovare le porzioni principali del codice costituenti i Model.

```
# TARGET TYPE MODEL

class TargetType(models.Model):
    """
    Target supported by Moon Cloud system
    """
    TYPES = (
        ('host', 'host'),
        ('windows', 'windows'),
        ('url', 'url'),
        ('azure', 'azure'),
        ('aws', 'aws')
    )
    name = models.CharField(max_length=150, choices=TYPES, default="host")
    descr = models.TextField(max_length=1000, default="none")  # Description of a target

    def __str__(self):
        return str(self.name)

    class Meta:
        ordering = ['id']


# CONTROL MODEL

class Control(MPTTModel):
    """
    Controls that can be part of Evaluations
    """
    other_id = models.IntegerField(default=-1, unique=True)
    parent = TreeForeignKey('self', on_delete=models.CASCADE, null=True, blank=True, related_name='
     children')
    name = models.CharField(max_length=150, unique=True)
    descr = models.TextField(max_length=1000, default="none")  # Description of a node in the
     taxonomy
    TYPES = (
        ('cat', 'category'),
        ('con', 'control')
    )
    # Possible node type of the taxonomy (category node or control node)
    node_type = models.CharField(max_length=3, choices=TYPES, default='cat')
    target_type = models.ForeignKey(TargetType, blank=True, null=True, on_delete=models.CASCADE)  #
     It's null for the root node and category nodes

    def __str__(self):
        return str(self.name)

    class MPTTMeta:
        level_attr = 'level'
        order_insertion_by = ['name']

    class Meta:
        ordering = ['tree_id', 'lft']


# EVALUATION MODEL

class Evaluation(MPTTModel):
    """
    Evaluation is composed by one or more Controls, and can be used by Users
    """
    other_id = models.IntegerField(default=-1, unique=True)
```

```
60      parent = TreeForeignKey('self', on_delete=models.CASCADE, null=True, blank=True, related_name='
         children')
61      name = models.CharField(max_length=150, unique=True)
62      descr = models.TextField(max_length=1000, default="none")  # Description of a node in the
         taxonomy
63      TYPES = (
64          ('cat', 'category'),
65          ('eva', 'evaluation')
66      )
67      # Possible node types of the taxonomy (category node or evaluation node)
68      node_type = models.CharField(max_length=3, choices=TYPES, default='cat')
69      controls = models.ManyToManyField(Control)  # Evaluation can be composed of one or more controls
70
71      def __str__(self):
72          return str(self.name)
73
74      class MPTTMeta:
75          level_attr = 'level'
76          order_insertion_by = ['name']
77
78      class Meta:
79          ordering = ['tree_id', 'lft']
80
81
82  # USER MODEL
83
84  class User(models.Model):
85      """
86      User registered to Moon Cloud with an email address, and can insert Target and launch Evaluations
87      """
88      other_id = models.IntegerField(default=-1, unique=True)
89      email = models.EmailField(max_length=50, unique=True)
90      evaluations = models.ManyToManyField(Evaluation, blank=True)  # Evaluations chosen by user
91
92      def __str__(self):
93          return str(self.email)
94
95      class Meta:
96          ordering = ['other_id', 'id']
97
98
99  # TARGET MODEL
100
101 class Target(models.Model):
102     """
103     Target (can be more than one) chosen by the user
104     """
105     user = models.ForeignKey(User, on_delete=models.CASCADE)  # User has chosen a target_type
106     other_id = models.IntegerField(default=-1, unique=True)
107     target_type = models.ForeignKey(TargetType, on_delete=models.CASCADE)  # TargetType Id
108
109     def __str__(self):
110         return str(self.user) + " " + str(self.other_id) + " " + str(self.target_type)
111
112     class Meta:
113         ordering = ['user']
```

Listing 4.1: Parti principali del codice che costituiscono i Model della soluzione.

A partire da questi Model vennero introdotte nel database le seguenti tabelle, le quali è possibile visionare nella Figura 4.1.

Control: contiene l'insieme dei software, o soltanto i riferimenti, che vengono poi effettivamente eseguiti all'interno di una Evaluation, i campi `other_id` (identificativo dell'Evaluation che fa riferimento al database effettivo di Moon Cloud), descr (una descrizione del funzionamento del controllo), `node_type` (definisce se il nodo è un Evaluation o una Categoria) definiscono le caratteristiche del controllo mentre lft, rght, `tree_id`, `level` e `parent` sono introdotti automaticamente dal package MPTT per poter rappresentare i dati in modo gerarchico, in-

fine `target_type_id` rappresenta, quel controllo a quale Target viene associato.

Evaluation: contiene l'insieme di Evaluation che un utente può eseguire per un certo Target, e allo stesso modo i campi contenuti nella tabella Control. La tabella intermedia *evaluation_ controls*permette di memorizzare quali Controlli sono associati a quali Evaluation.

User: contiene gli utenti registrati alla piattaforma Moon Cloud, e sono anche loro, come con le tabelle precedenti, identificati con un campo `other_id`, e distinti da un email. La tabella intermedia *user_ evaluations* permette di memorizzare quali Evaluation un utente ha selezionato e usato.

Target: contiene i Target (o asset) un utente ha inserito e sui quali vuole effettuare dei processi di monitoraggio e verifica, attraverso l'applicazione di politiche e Evaluation.

TargetType: contiene i tipi di Target supportati da Moon Cloud. In generale sono supportati: *URL*, rappresentante l'URL dell'applicativo web, *Host*, viene specificato l'indirizzo IP identificativo di un host e si distingue tra sistema operativo *Windows* o *Linux* eseguito su esso, *Aws*, rappresentante il servizio di cloud computing del gruppo Amazon, e *Azure*, definisce un servizio cloud fornito da Microsoft.

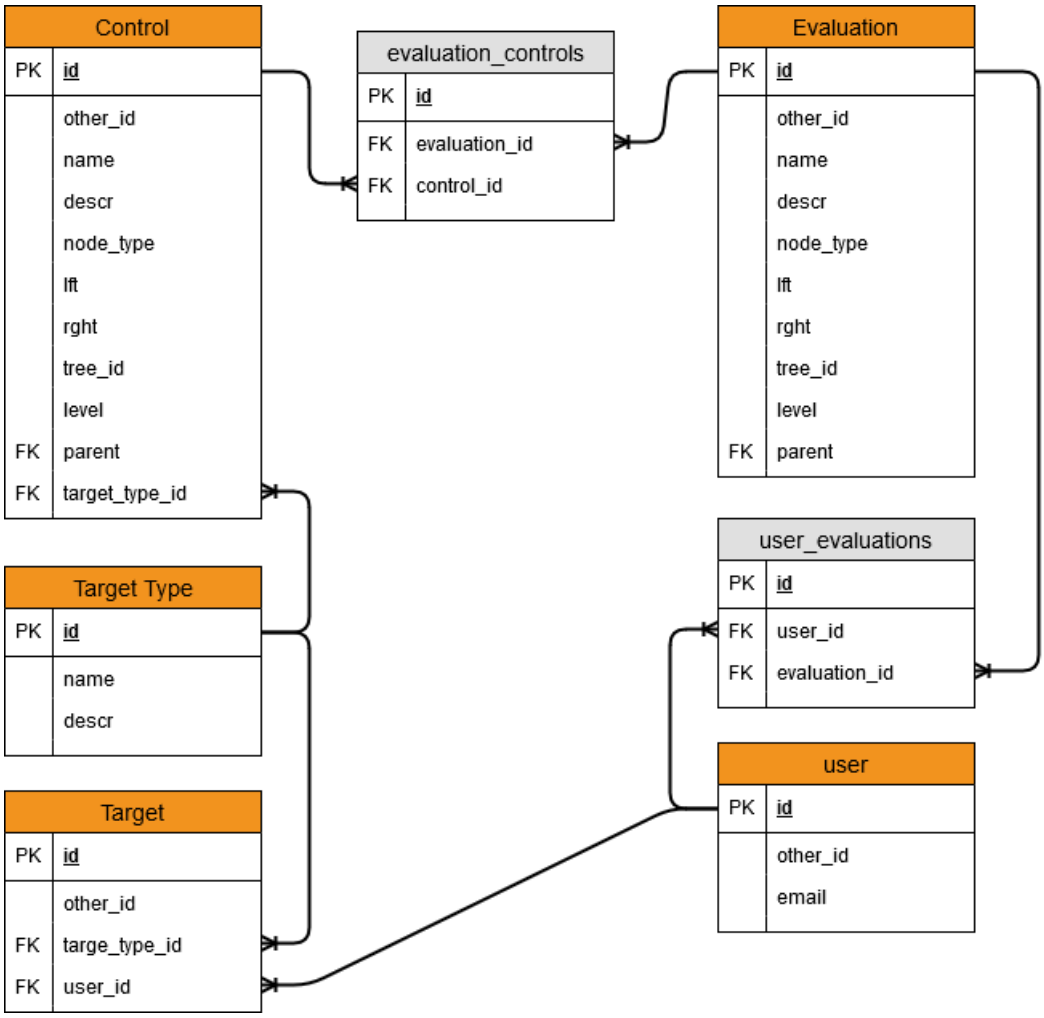

Figura 4.1: Struttura del database.

Realizzazione delle View

Successivamente per poter testare che la tassonomia creata per le Evalution e i Controlli fosse corretta e funzionante si è implementata un'interfaccia Web a scopo didattico. Avviando il server, viene mostrata una home page la quale contiene una barra di navigazione da cui è possibile accedere, attraverso il *Admin*, alla admin page offerta da Django (successivamente personalizzata, per poter manipolare la base di dati, e aggiungere, eliminare item o utenti), mentre tramite il link *Home* è possibile tornare a questa pagina. Il contenuto di questa pagina mostra una breve descrizione di un sistema di raccomandazione, e accedere tramite i due appositi pulsanti alle pagine specifiche per

la navigazione della tassonomia delle Evaluation piuttosto che dei Controlli; tutto questo viene mostrato dalla Figura 4.2 e il relativo Listing 4.2.

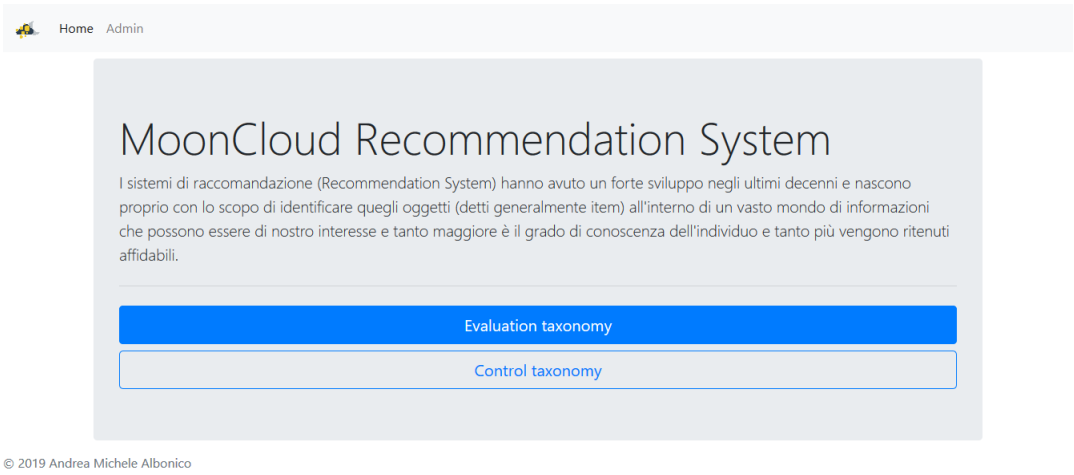

Figura 4.2: Home page dell'applicativo web a scopo didattico.

```
def index(request):
    """
    Index page where you can choose to navigate the evaluation taxonomy or the control taxonomy.
    :param request: HTTP request
    :return: HTTP response with the template to show to the user
    """
    return render(request, "recommendation_app/index.html")
```

Listing 4.2: Parte principale del codice delle View della soluzione per gestire l'accesso alla home page.

Una volta scelta la tassonomia, di Controlli o delle Evaluation, su cui si vuole navigare, viene mostrata una pagina dove nella metà a sinistra troviamo la possibilità di selezionare un nodo della tassonomia e l'operazione che si vuole svolgere su quel nodo, inoltre, è possibile effettuare una richiesta di generazione di file in formato e linguaggio DOT e relativa immagine in formato .png, attraverso il pulsante *Request schema*. Proseguendo più in basso, vengono mostrati tutti i nodi a cui è stato associato il tipo Categoria e il tipo Evaluation. Nella porzione destra della pagina web viene mostrata l'intera struttura della tassonomia e un link da quale è possibile accedere a una pagina di dettaglio che mostra una simile a quella presente sul database. Questa pagine è visibile dalla Figura 4.3 e dal Listing 4.3.

```
def tax_index(request, taxonomy_used):
    """
    The home page shows all taxonomy and a form to make operations on it.
    :param request: HTTP request
    :param taxonomy_used: specify if it's used the Control taxonomy or the Evaluation taxonomy
    :return: HTTP response with the template to show to the user
```

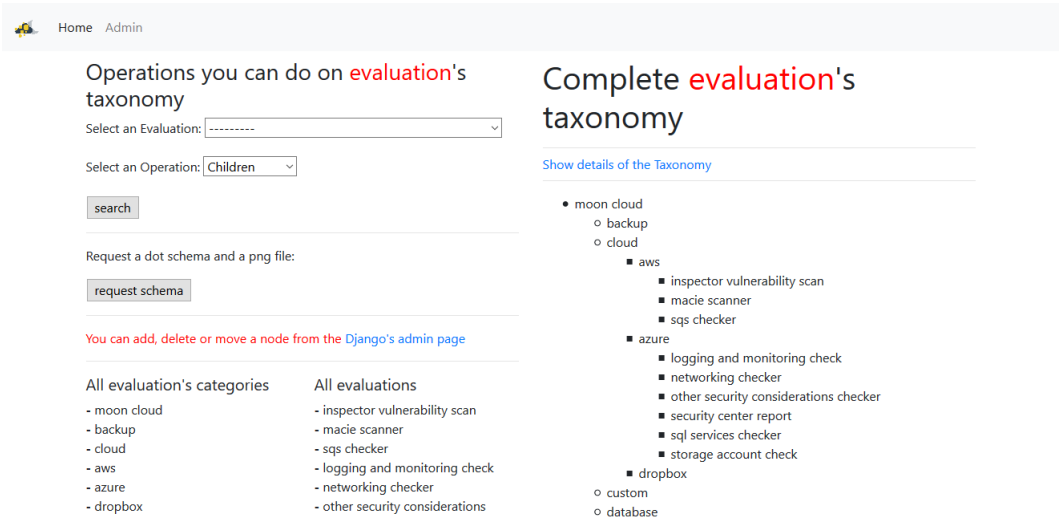

Figura 4.3: Home page per la navigazione della tassonomia delle Evaluation.

```
    """
    # If this is a POST request we need to process the form data
    if request.method == 'POST':
        # Create a form instance and populate it with data from depending on the taxonomy_used
        if (taxonomy_used == "evaluation"):
            form = EvaluationOperationForm(request.POST)
        else:
            form = ControlEvaluationForm(request.POST)
        # Check whether it's valid:
        if form.is_valid():
            # Process the data in form.cleaned_data as required
            nodename_form = form.cleaned_data['nodeName']
            taxonomy_operation_form = form.cleaned_data['actionTax']
            # Redirect to a new URL (page that show a part of the taxonomy, depending on the action
     user has chosen):
            return redirect(
                reverse('rec:tax_index', args=[taxonomy_used]) + str(nodename_form) + '_' +
     taxonomy_operation_form)
    # If id's a GET method we'll create a blank form
    else:
        if (taxonomy_used == "evaluation"):
            form = EvaluationOperationForm()
        else:
            form = ControlEvaluationForm()

    # Depending on the taxonomy_used, I'm getting all the categories of Evaluations or Controls
     taxonomy and save it in a
    # list called "categories_list"
    if (taxonomy_used == "evaluation"):
        q_categories = Evaluation.objects.filter(node_type='cat')
    else:
        q_categories = Control.objects.filter(node_type='cat')
    categories_list = []
    for node in q_categories:
        categories_list.append(node.name)

    # Depending on the taxonomy_used, I'm getting all the categories of Evaluations or Controls node
     in the taxonomy
    # and save it in a list called "node_list"
    if (taxonomy_used == "evaluation"):
        q_nodes = Evaluation.objects.filter(node_type='eva')
    else:
        q_nodes = Control.objects.filter(node_type='con')
    node_list = []
    for node in q_nodes:
        node_list.append(node.name)

    # Depending on the taxonomy_used, I'm getting all the Evaluations or Controls taxonomy
    if (taxonomy_used == "evaluation"):
```

```
52          tax = Evaluation.objects.all()
53      else:
54          tax = Control.objects.all()
55
56      # Passing the complete taxonomy and data to fill the form so you can operate on the taxonomy
57      args = {'tax': tax,
58              'categories': categories_list,
59              'nodes': node_list,
60              'form': form,
61              'request_path': taxonomy_used}
62
63      return render(request, "recommendation_app/tax_index.html", args)
```

Listing 4.3: Home page dalla quale è possibile richiamare diverse operazioni eseguibili sulla tassonomia.

Le attività che è possibile svolgere sulla tassonomia, sia dei Controlli sia delle Evaluation, sono le seguenti, mentre le operazioni di manipolazione dei dati memorizzati dal database vengono svolte con l'ausilio della admin page messa a disposizione da Django.

- Per ogni singolo nodo è possibile recuperare: i discendenti, i figli, la famiglia o i fratelli. Si ottiene un risultato come nella Figura 4.4 nel caso in cui si è scelti l'*Evaluation Http robustness check* e si vuole ritornare tutta la famiglia di quel nodo.

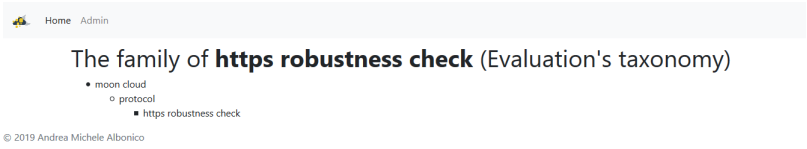

Figura 4.4: Risultato dell'operazione selezionata sul nodo in questione.

```
1       # Methods to navigate the taxonomy
2
3       def show_descendants(request, nodename, taxonomy_used):
4           """
5           Based on the MPTT's method 'get descendants' that return the descendants of a model
        instance, in tree order
6           :param request: HTTP request
7           :param nodename: name (it's unique for each node) of a node in the taxonomy
8           :param taxonomy_used: specify if it's used the Control taxonomy or the Evaluation
        taxonomy
9           :return: HTTP response with the template to show to the user
10          """
11          if (taxonomy_used == 'evaluation'):
12              q_result = Evaluation.objects.get(name=nodename).get_descendants(include_self=
        False)
13              # Get the count of descendants of the model instance
14              q_result_num = Evaluation.objects.get(name=nodename).get_descendant_count()
15          else:
16              q_result = Control.objects.get(name=nodename).get_descendants(include_self=False)
17              # Get the count of descendants of the model instance
18              q_result_num = Control.objects.get(name=nodename).get_descendant_count()
19
20          return render(request, "recommendation_app/tax_node_details.html",
21                        {'tax_type': (str(taxonomy_used)).capitalize(),
22                        'descendants': q_result,
23                        'node_exe': nodename,
24                        'method': 'descendants',
25                        'num_descendants': q_result_num})
26
```

```
27
28      def show_children(request, nodename, taxonomy_used):
29          """
30          Based on the MPTT's method 'get children' that return the immediate children of a
        model instance, in tree order
31          :param request: HTTP request
32          :param nodename: name (it's unique for each node) of a node in the taxonomy
33          :param taxonomy_used: specify if it's used the Control taxonomy or the Evaluation
        taxonomy
34          :return: HTTP response with the template to show to the user
35          """
36          if (taxonomy_used == 'evaluation'):
37              q_result = Evaluation.objects.get(name=nodename).get_children()
38          else:
39              q_result = Control.objects.get(name=nodename).get_children()
40
41          return render(request, "recommendation_app/tax_node_details.html",
42                        {'tax_type': (str(taxonomy_used)).capitalize(),
43                        'children': q_result,
44                        'node_exe': nodename,
45                        'method': 'children'})
46
47
48      def show_family(request, nodename, taxonomy_used):
49          """
50          Based on the MPTT's method 'get family' that return the ancestors, the model instance
        itself and the descendants,
51          in tree order
52          :param request: HTTP request
53          :param nodename: name (it's unique for each node) of a node in the taxonomy
54          :param taxonomy_used: specify if it's used the Control taxonomy or the Evaluation
        taxonomy
55          :return: HTTP response with the template to show to the user
56          """
57          if (taxonomy_used == 'evaluation'):
58              q_result = Evaluation.objects.get(name=nodename).get_family()
59          else:
60              q_result = Control.objects.get(name=nodename).get_family()
61
62          return render(request, "recommendation_app/tax_node_details.html",
63                        {'tax_type': (str(taxonomy_used)).capitalize(),
64                        'family': q_result,
65                        'node_exe': nodename,
66                        'method': 'family'})
67
68
69      def show_siblings(request, nodename, taxonomy_used):
70          """
71          Based on the MPTT's method 'get siblings' that return siblings of the model instance (
        root nodes are considered
72          to be siblings of other root nodes)
73          :param request: HTTP request
74          :param nodename: name (it's unique for each node) of a node in the taxonomy
75          :param taxonomy_used: specify if it's used the Control taxonomy or the Evaluation
        taxonomy
76          :return: HTTP response with the template to show to the user
77          """
78          if (taxonomy_used == 'evaluation'):
79              q_result = Evaluation.objects.get(name=nodename).get_siblings()
80          else:
81              q_result = Control.objects.get(name=nodename).get_siblings()
82
83          return render(request, "recommendation_app/tax_node_details.html",
84                        {'tax_type': (str(taxonomy_used)).capitalize(),
85                        'siblings': q_result,
86                        'node_exe': nodename,
87                        'method': 'siblings'})
88
```

Listing 4.4: Codice utilizzato all'interno delle View per implementare le operazioni per restituire i discendenti, i figli, la famiglia o i fratelli.

- Attraverso il pulsante *request schema* nella home page della tassonomia si può effettuare una richiesta al software Graphviz il quale attraverso un file scritto in linguaggio DOT genera lo schema della tassonomia sotto forma d'immagine in formato .png.

Il DOT è un linguaggio descrittivo per grafi, tipicamente i file hanno estensione .gv o .dot, che in combinazione con il software Graphviz è possibile effettuare il rendering della sintassi DOT in un immagine più significativa. Un esempio di tale linguaggio è mostrato nel Listing 4.5 viene mostrato un esempio di scrittura con linguaggio DOT.

```
graph {
    1 [label="moon cloud"]
    12 [label="protocol"]
    1 -- 12
    11 [label="infrastructure"]
    1 -- 11
    4 [label="cloud"]
    1 -- 4
    16 [label="backup"]
    1 -- 16
    2 [label="database"]
    1 -- 2
    13 [label="web"]
    1 -- 13
    9 [label="human"]
    1 -- 9
    8 [label="custom"]
    1 -- 8
    3 [label="operating system"]
}

```

Listing 4.5: Codice parziale utilizzato per realizzare lo schema della tassonomia per le Evaluation.

```
def dot_graph(request, taxonomy_used):
    """
    Create the .dot file (based on the Dot language) and the graph showing the taxonomy in .png format
    :param request: HTTP request
    :param taxonomy_used: specify if it's used the Control taxonomy or the Evaluation taxonomy
    :return: HTTP response with the template to show to the user
    """
    # Create a graph object
    taxonomy_dot_object = Graph(comment='Taxonomy', format='png')
    # Fill the graph with every node in the database (evaluations/controls node and categories nodes),
    # and create a link with the parent node
    if (taxonomy_used == 'evaluation'):
        taxonomy_nodes = Evaluation.objects.all()
    else:
        taxonomy_nodes = Control.objects.all()
    i = 0
    for node in taxonomy_nodes.order_by('level'):
        # This If construct will prevent the adding of an empty node to the root node in the graph
        if (i == 0):
            # Insert the root node
            taxonomy_dot_object.node(str(node.id), label=str(node.name))
        else:
            # Insert the other nodes
            taxonomy_dot_object.node(str(node.id), label=str(node.name))
            taxonomy_dot_object.edge(str(node.parent_id), str(node.id))
        i += 1
    # Specify where I want to save the .png image and the .dot file
    taxonomy_dot_object.render('taxonomy_output/taxonomy.dot')

    # This function is used to zip a directory
    def make_zipdir(path, ziph):
        # Ziph is zipfile handle
        for root, dirs, files in os.walk(path):
            for file in files:
                ziph.write(os.path.join(root, file))

    # Making the zip file
    zip_file = zipfile.ZipFile('taxonomy_output.zip', 'w', compression=zipfile.ZIP_DEFLATED)
    make_zipdir('taxonomy_output/', zip_file)
    zip_file.close()

```

```
42          # Remove the directory which was zipped and all files inside
43          shutil.rmtree("taxonomy_output/")
44
45          return redirect(reverse('rec:index'))
46
```

Listing 4.6: Codice utilizzato per la realizzazione del file .dot e relativa immagine .png.

- Inoltre è possibile osservare in maniera più approfondita le informazioni rilevanti sulla tassonomia contenute nelle tabelle del database; e nel caso, della tassonomia delle Evaluation, è possibile simulare il processo di Item Recommendation, premendo il relativo tasto rispetto al nodo su cui si vuole determinare le altre Evaluation simili.

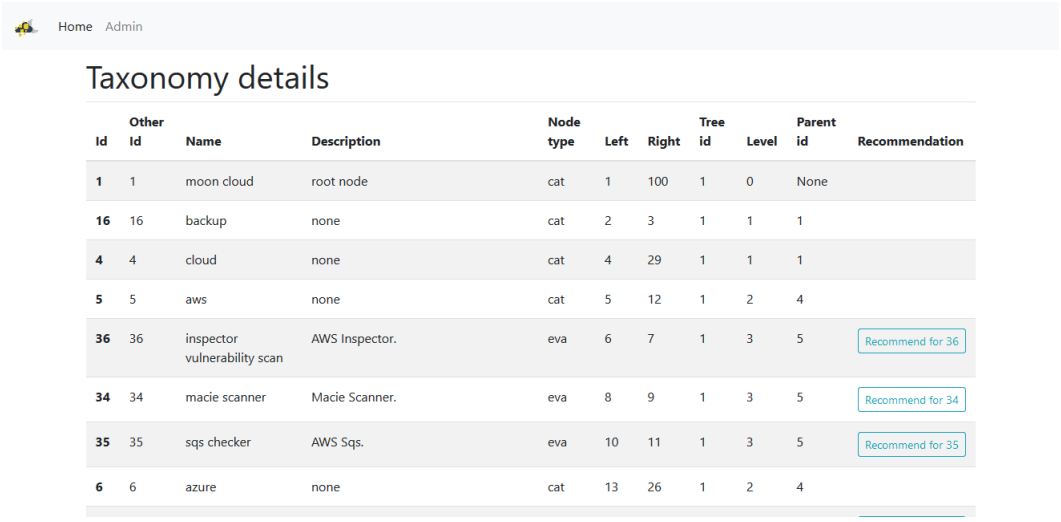

Figura 4.5: Dettagli della tassonomia sotto forma di tabella come nella base di dati.

```
1   def tax_details(request, taxonomy_used):
2       """
3       Show the taxonomy's details page showing an overview of the taxonomy
4       :param request: HTTP request
5       :param taxonomy_used: specify if it's used the Control taxonomy or the Evaluation
        taxonomy
6       :return: HTTP response with the template to show to the user
7       """
8       if (taxonomy_used == 'evaluation'):
9           tax_details_obj = Evaluation.objects.all()
10      else:
11          tax_details_obj = Control.objects.all()
12
13      return render(request, "recommendation_app/tax_details.html",
14                    {'tax_details': tax_details_obj,
15                     'taxonomy_used': taxonomy_used})
16
```

Listing 4.7: Codice utilizzato per la realizzazione della View che implementa la pagina web del dettaglio della Tassonomia.

View per i processi di raccomandazione

In generale gli algoritmi di raccomandazione implementati in questo progetto, e descritti nel Capitolo 3, vengono richiamati e utilizzati come API REST, un tipo di architettura basata sul protocollo HTTP. Un sistema REST, per funzionare, prevede una struttura degli URL ben definita (atta a identificare univocamente una risorsa o un insieme di risorse) l'utilizzo dei verbi HTTP specifici per il recupero d'informazioni (GET), per la modifica (POST, PUT, PATCH, DELETE) e per altri scopi (OPTIONS, ecc.).
In questo caso a ogni URL è associata una azione particolare la quale va a richiamare la View che permette di determinare un insieme di raccomandazioni sulla base di un parametro in ingresso, come è mostrato nel Listing 4.8.

```
# RECOMMENDATION VIEWS

# Item recommendation process
# Depending on the <str:item_other_id> value I decide on which evaluation of the taxonomy I want to recommend on
path('recommendation/item/<str:item_other_id>/', recommendation_views.item_recommendation,
        name='item_recommendation'),

# User recommendation process
# Depending on the <str:user_other_id> value I decide on which user I want to recommend on
path('recommendation/user/<str:user_other_id>/', recommendation_views.user_recommendation,
        name='user_recommendation'),

# Hybrid recommendation process (User and Item recommendation process)
# Depending on the <str:User_other_id> value I decide on which user I want to recommend on with an hybrid approach
path('recommendation/hybrid/<str:user_other_id>/', recommendation_views.hybrid_recommendation,
        name='hybrid_ui_recommendation'),

# Target recommendation process
# Depending on the <str:target_type_id> value I decide on which user I want to recommend on
path('recommendation/target/<str:target_type_id>/', recommendation_views.target_recommendation,
        name='target_recommendation'),
```

Listing 4.8: Porzione parziale del codice contenuto nell'URL mapper per implementare i sistemi di raccomandazione.

Sulla base dell'URL che viene richiamato e sul parametro (come `user_other_id`, `item_other_id`, ecc.) verranno fornite le relative raccomandazioni. Nel caso di questo progetto, le View che possono essere richiamate permettono soltanto chiamate HTTP con metodo GET e restituiscono una risposta HTTP in formato JSON, queste risposte contengono una lista di dizionari e ciascuno contiene l'`other_id` della Evaluation che si vuole raccomandare. Di seguito nel Listing 4.9 vengono mostrati le porzioni di codice rappresentanti le View per gli algoritmi di raccomandazione per Item (in questo caso Evaluaiton), Utenti e Target. Inoltre nel Listing 4.10 è possibile osservare alcuni esempi e risposte di richieste HTTP a questi algortmi.

```
# Item recommendation API REST
@api_view(['GET'])
def item_recommendation(request, item_other_id):
    """
```

```
    Returns recommendations for the using an item-based recommendation algorithm for the Evaluation `
     item_other_id`

    :param request: http GET request
    :param item_other_id: value representing the other_id of an Item (Evaluation)
    :return: json response with the evaluations to recommend
    """
    # Trying to retrieve the actual node with item_other_id
    item = Evaluation.objects.get(other_id=item_other_id)

    similar_item_evaluations = item_recommendation_alg(item_other_id)
    # Cleaning the data, deleting all the keys except 'other_id'
    similar_item_evaluations_serilized = EvaluationSerializerRecommendation(similar_item_evaluations,
      many=True).data

    return JsonResponse(similar_item_evaluations_serilized, safe=False)

# User recommendation API REST
@api_view(['GET'])
def user_recommendation(request, user_other_id):
    """
    Returns recommendations for the using an user-based recommendation algorithm for the user `
     user_other_id`

    :param request: http GET request
    :param user_other_id: value representing the other_id of a User
    :return: json response with the evaluations to recommend
    """
    # Trying to retrieve the actual User with user_other_id
    user = User.objects.get(other_id=user_other_id)

    target_user_evaluations, similar_user_evaluations = user_recommendation_alg(user_other_id)
    # Cleaning the data, deleting all the keys except 'other_id'
    similar_user_evaluations_serilized = EvaluationSerializerRecommendation(similar_user_evaluations,
      many=True).data

    return JsonResponse(similar_user_evaluations_serilized, safe=False)

# Target Recommendation API REST
@api_view(['GET'])
def target_recommendation(request, target_type_id):
    """
    For a target (Target_Type like 'host', 'aws', 'url', etc.) chose by a user this algorithm search
     the possible
    evaluations that can be recommended for that user.

    :param request: http GET request
    :param target_type_id: identifier of a particular target_type
    :return: json response with the evaluations to recommend
    """
    # Trying to retrieve the actual Target Type, with id equal to target_type_id, chosed by a user
    target = TargetType.objects.get(id=target_type_id)

    target_evaluations = target_recommendation_alg(target_type_id)

    return JsonResponse(target_evaluations, safe=False)
```

Listing 4.9: Porzione parziale del codice contenuto nelle View per implementare i sistemi di raccomandazione.

```
Esempio di chiamata per lo Item Recommendation Algorithm
URL: http://127.0.0.1:8000/recommendation/item/34/
HTTP method: GET

Esempio di risposta per lo Item Recommendation Algorithm
HTTP Status: 200 OK
HTTP Response Body:
[
    {"other_id": 35},
    {"other_id": 36}
]

Esempio di chiamata per lo User Recommendation Algorithm
URL: http://127.0.0.1:8000/recommendation/user/10/
HTTP method: GET

Esempio di risposta per lo User Recommendation Algorithm
HTTP Status: 200 OK
HTTP Response Body:
[
    {"other_id": 36}
```

```
]

Esempio di chiamata per lo Hybrid Recommendation Algorithm
URL: http://127.0.0.1:8000/recommendation/hybrid/10/
HTTP method: GET

Esempio di risposta per lo Hybrid Recommendation Algorithm
HTTP Status: 200 OK
HTTP Response Body:
[
    {"other_id": 23},
    {"other_id": 24},
    {"other_id": 25},
    {"other_id": 26},
    {"other_id": 28},
    {"other_id": 29},
    {"other_id": 30},
    {"other_id": 31},
    {"other_id": 32},
    {"other_id": 35},
    {"other_id": 36},
    {"other_id": 43},
    {"other_id": 45},
    {"other_id": 46},
    {"other_id": 47},
    {"other_id": 48}
]

Esempio di chiamata per lo Target Recommendation Algorithm
URL: http://127.0.0.1:8000/recommendation/target/1/
HTTP method: GET

Esempio di risposta per lo Target Recommendation Algorithm
HTTP Status: 200 OK
HTTP Response Body:
[
    {"other_id": 25},
    {"other_id": 29},
    {"other_id": 30},
    {"other_id": 24},
    {"other_id": 28},
    {"other_id": 31},
    {"other_id": 23},
    {"other_id": 26}
]
```

Listing 4.10: Esempi di chiamate e risposte HTTP per i diversi algoritmi di raccomandazione.

Personalizzazione Admin Page

Per poter agilmente manipolare la base di dati, senza dover scrivere manualmente le query in puro Sql, Django mette a disposizione la cosiddetta Admin Page mostrata in Figura 4.6, che è stata personalizzata per mostrare le tabelle su cui è possibile effetuare modifiche, e per ognuna vengono visualizzate le colonne più rilevanti, come mostrato dalla Figura 4.7 nel caso della tabella Evaluation, e dalla quale è possibile effettuare ricerche, attraverso l'apposito campo, eliminare direttamente i record contenuti nel database e aggiungerne dei nuovi, come mostrato in Figura 4.8.
In particolare in questo ultimo caso, sulla base di come è stato costruito il Model della relativa tabella, si potranno avere dei campi la cui compilazione è opzionale e altri in cui è obbligatorio compilare per la creazione del record.

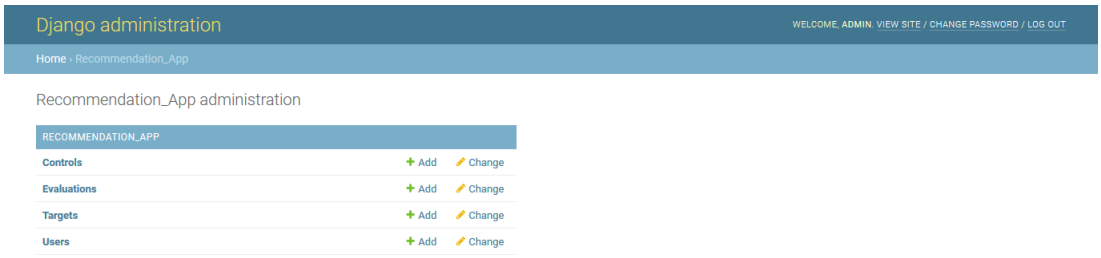

Figura 4.6: Admin page creata automaticamente da Django.

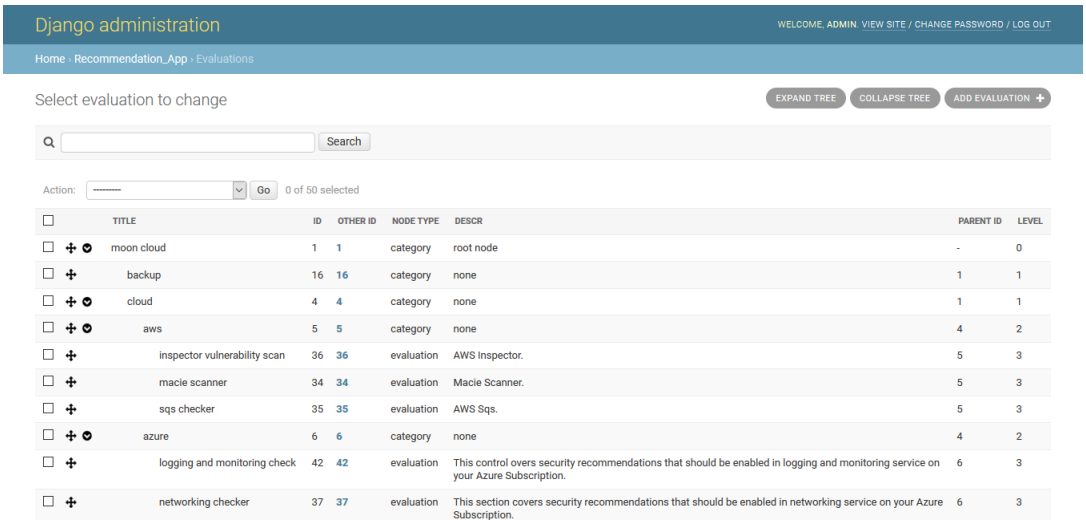

Figura 4.7: Esempio di Admin page per la tabella delle Evaluation.

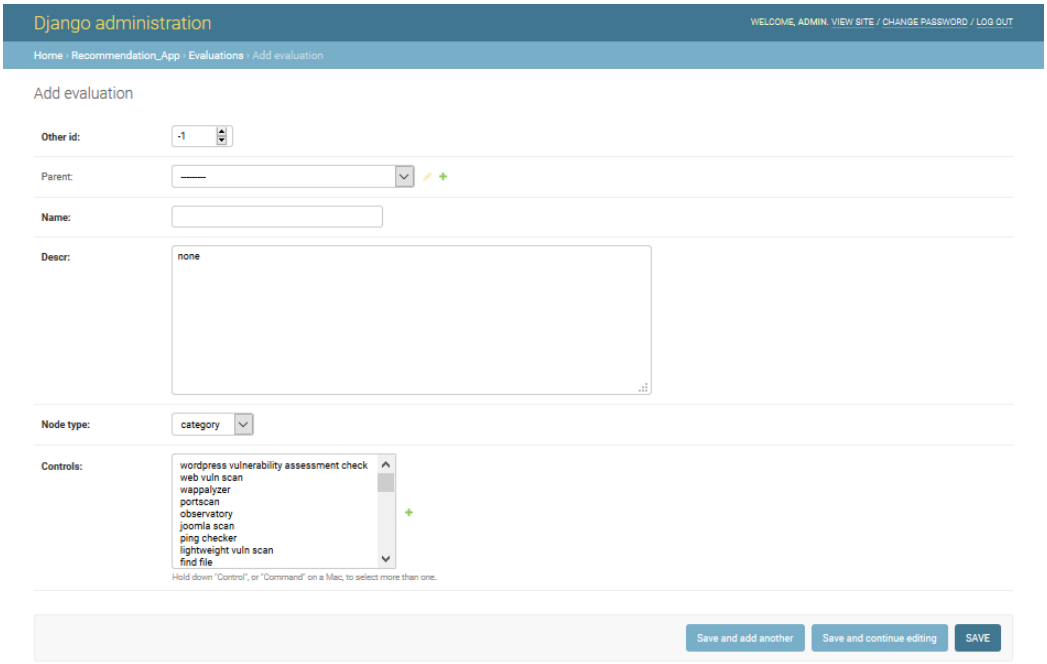

Figura 4.8: Esempio di Admin page per il caso in cui si vuole aggiungere una nuova Evaluation.

Consistenza tra i database

Per fare in modo che il database creato in questo progetto e l'attuale database utilizzato dalla piattaforma di Moon Cloud fossero consistenti, quindi le

informazioni presenti nell'uno siano identiche a quelle nell'altro, si è pensato d'implementare un sistema di API REST.
Nel caso vengano fatte modifiche, aggiunte o cancellazioni di dati relativi a Controlli, Evaluation, Target, User o Target Type è possibile richiamare tramite appositi URL, mostrati nel Listing 4.11, queste funzioni ed eseguire la relativa operazione sul database implmentato nella soluzione, senza dover accedere e manipolare direttamente la base di dati.

```
# API REST EXTERNAL to maintain database consistency
# Control views
url(r'^control/$', control_views.ControlListCreate.as_view(),
    name="control_list_create_view"),
url(r'^control/(?P<other_id>[0-9]+)/$', control_views.ControlRetrieveDeleteUpdate.as_view(),
    name="control_retrieve_delete_update_view"),

# Evaluation views
url(r'^evaluation/$', evaluation_views.EvaluationListCreate.as_view(),
    name="evaluation_list_create_view"),
url(r'^evaluation/(?P<other_id>[0-9]+)/$', evaluation_views.EvaluationRetrieveDeleteUpdate.as_view(),
    name="evaluation_retrieve_delete_update_view"),

# Target views
url(r'^target/$', target_views.TargetListCreate.as_view(),
    name="target_list_create_view"),
url(r'^target/(?P<other_id>[0-9]+)/$', target_views.TargetRetrieveDelete.as_view(),
    name="target_retrieve_delete_view"),

# User views
url(r'^user/$', user_views.UserListCreate.as_view(),
    name="user_list_create_view"),
url(r'^user/(?P<other_id>[0-9]+)/$', user_views.UserRetrieveDeleteUpdate.as_view(),
    name="user_retrieve_delete_update_view"),
url(r'^user_evaluations/$', user_views.UserEvaluationUpdate.as_view(),
    name="user_evaluations_update_view"),
```

Listing 4.11: Porzione di codice dell'URL Mapper contenti le URL usate per mantenere la consistenza dei dati.

Implementazione in Docker

Una volta preparata la base e gli algoritmi di raccomandazione, con tutto ciò a essi collegati (View, URL, API REST); l'ultima fase nel ciclo di sviluppo è il deployment e la sua preparazione. In questo caso l'app viene deployata, come il resto dei componenti di Moon Cloud, in una serie di Container Docker. Per tale ragione, è stato predisposto il file *docker-compose.yml*, il quale viene usato dalla Docker Machine per poter creare un Immagine contenente il necessario per l'esecuzione all'interno del Container, oltre alle caratteristiche del Container stesso e delle informazioni usate per poter comunicare con l'esterno.

```
    version: '3'

    services:
      db:
        networks:
          - mooncloud
        image: postgres:12.1
        restart: always
        volumes:
          - "./init_db/initdb.sh:/docker-entrypoint-initdb.d/1-initdb.sh"
```

```
11        environment:
12          POSTGRES_USER: postgres
13          POSTGRES_PASSWORD: postgres
14
15      api:
16        networks:
17          - mooncloud
18        build:
19            dockerfile: Dockerfile
20            context: .
21        command: ./entrypoint.sh
22        ports:
23          - "8000:8000"
24        depends_on:
25          - db
26        environment:
27          DATABASE_HOST: db
28          DATABASE_PORT: 5432
29          DATABASE_USER: postgres
30          DATABASE_PASSWORD: postgres
31          DATABASE_NAME: mooncloud_rec
32
33    networks:
34      mooncloud:
35        external: true
```

Listing 4.12: Contenuto del file docker-compose.yml per il deployment del sistema di raccomandazione all'interno di un Container.

Questo file permette di configurare i servizi della propria applicazione; i quali, attraverso un singolo comando, possono essere creati e inizializzati a partire da questo singolo file. La prima riga specifica sempre la versione (*version*) del formato del file di Compose, successivamente si definiscono i *services* e le *networks*. In questo progetto si è fatto uso di due servizi (*db* e *api*), i quali speficiano le proprietà dei rispettivi Container uno per implementare il database (*db*) e il secondo per le API REST (*api*); nel primo caso per il processo di *build* si fa uso di un'Immagine che è già stata pubblicata in un Docker Registry, mentre nel secondo caso essa viene costruita a partire dal codice sorgente scritto nel suo *Dockerfile*. Per fare in modo che servizi siano accessibili dall'esterno il numero di porta (*ports*) deve essere indicato, nel caso del sevizio *api* ciò è specificato all'interno del *Dockerfile*.
Le *networks* definiscono le regole di comunicazione tra i Container, e tra i Container e gli host; delle reti in comune permettono a diversi servizi di essere visibili l'uno con l'altro.

Ogni qualvolta è stata portata a termine una fase dell'implementazione, proposta in questa tesi, sono state predisposte dei processi di verifica del funzionamento del codice scritto fino a quel momento. Questo genere di test, vennero fatti per essere certi che le varie fasi fossero in grado di funzionare insieme.
I primi test sono fatti sul database, accertandosi del suo funzionamento e di quello del package MPTT usato per costruire la tassonomia delle Evaluation

e quella dei Controlli. Successivamente vennero fatte delle verifiche sulla corretta interazione, con la base di dati, attraverso le View a scopo didattico e che le modifiche fatte alle admin page standard di Django dessero i risultati attesi. Una volta implementati gli algoritmi di raccomandazione, e relative View, e il sistema di API REST per il mantenimento della consistenza tra il database, creato in questa soluzione, e quello attualmente in uso su Moon Cloud, venne creato un sistema di test automatizzato, grazie alle funzioni built-in di Django, in cui ogni test richiama la funzione da testare, salva il suo risultato, e lo confronta con quello atteso, oltre a verificare che il codice della risposta HTTP corrisponda a quello atteso.
Per semplificare lo sviluppo e la verifica del corretto funzionamento si è voluto creare dei test aggiuntivi tramite l'ausilio di Postman; una piattaforma nata per lo sviluppo e il test di API, inoltre semplifica molti step per il loro sviluppo così da poter creare API migliori e più velocemente. Inoltre permette di generare una documentazione machine-readable così da rendere le API più facili da usare.

Capitolo 5

Conclusioni

La soluzione proposta in questa tesi vuole introdurre un sistema di raccomandazione in un mondo in cui spesso non vengono introdotti perché popolato da utenti esperti che non ne avrebbero bisogno; invece con il progetto qui proposto si darebbe una possibilità a un maggior numero di utenti di accedere a servizi su un sistema Cloud di Security Assurance, come Moon Cloud, in totale sicurezza e affidabilità.
Con questo lavoro è stato possibile studiare e approfondire il linguaggio di programmazione Python, unitamente al framework Django per la realizzazione di applicativi web e la tecnologia Docker per il rilascio in ambienti isolati e indipendenti di software; inoltre sono stati approfonditi i temi legati ai Recommendation System e al mondo del machine learning.

5.1 Sviluppi futuri

Il sistema di raccomandazione sviluppato in questo tesi offre delle raccomandazioni di tipo basico, tuttavia è in grado di supportare gli utenti nell'utilizzo della piattaforma Moon Cloud. Offre altresì spunti di miglioramento, ad esempio l'introduzione di un sistema di valutazione delle Evaluation o dei Controlli da parte dell'utente così da incrementare la precisione del sistema di raccomandazione, il quale terrebbe conto anche di queste valutazioni.

Bibliografia

[1] M. Anisetti et al. «A semi-automatic and trustworthy scheme for continuous cloud service certification». In: *IEEE TRANSACTIONS ON SERVICES COMPUTING* (2017). DOI: 10.1109/TSC.2017.2657505.

[2] M. Anisetti et al. «Moon Cloud: A Cloud Platform for ICT Security Governance». In: (dic. 2018), pp. 1–7. DOI: 10.1109/GLOCOM.2018.8647247.

[3] *Django documentation*. https://docs.djangoproject.com/en/2.2/. 2019.

[4] Minh-Phung Do, Dung Nguyen e Academic Network of Loc Nguyen. «Model-based approach for Collaborative Filtering». In: ago. 2010.

[5] *MDN Django documentation*. https://developer.mozilla.org/it/docs/Learn/Server-side/Django/. 2019.

[6] Miquel Montaner, Beatriz López e Josep Lluís de la Rosa. «A Taxonomy of Recommender Agents on the Internet». In: *Artificial Intelligence Review* 19.4 (giu. 2003), pp. 285–330. ISSN: 1573-7462. DOI: 10.1023/A:1022850703159. URL: https://doi.org/10.1023/A:1022850703159.

[7] *Python 3.7 documentation*. https://docs.python.org/3.7/. 2019.

[8] Badrul Sarwar et al. «Item-based Collaborative Filtering Recommendation Algorithms». In: WWW '01 (2001), pp. 285–295. DOI: 10.1145/371920.372071. URL: http://doi.acm.org/10.1145/371920.372071.