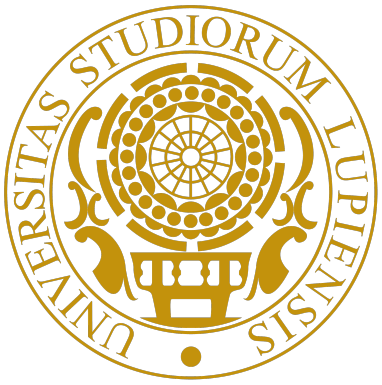

UNIVERSITÀ DEL SALENTO

Dipartimento di Ingegneria dell'Innovazione
Corso di Laurea in Ingegneria Informatica
Corso di Gestione di Big Data

REMOTE: Progettazione e Sviluppo di un sistema di gestione di dati a supporto del REndering e MOnitoraggio delle strutTurE

Manuale tecnico

Supervisor:

Prof. Longo Antonella

Studenti:

Caforio Mirko

Barone Andrea

Anno accademico 2023/2024

Indice

Introduzione

1.1 Premessa

L'obiettivo di questo progetto è quello di progettare e realizzare un sistema per supportare la visualizzazione 3D di una qualsiasi opera d'arte (ad esempio un ponte, un edificio, ecc...) e la simulazione di parametri importanti, legati al monitoraggio dell'opera d'arte stessa, come ad esempio la temperatura, l'umidità, la pressione o altro. Il sistema alla base, prevede anche la progettazione e lo sviluppo di un sistema che permetta la gestione dei dati inerenti l'opera d'arte da trattare e i dati ottenuti dalla simulazione dei sensori inseriti all'interno della struttura utili al monitoraggio e per eventuali analisi sui dati raccolti.

1.2 Giustificazione di questo lavoro

La nascita di questo progetto, è legata all'obiettivo di voler sfruttare i file *IFC* (Industry Foundation Classes) per ottenere una rappresentazione 3D più completa possibile del modello rappresentato dal file stesso. Oltre a questo, c'è la necessità di ottenere un modello dati che rappresenti completamente l'opera d'arte rappresentata dal file *IFC*, per poter immagazzinare tutti le informazioni riguardanti quest'ultima. Di conseguenza, una volta acquisiti questi dati, si vuole simulare il comportamento di sensori all'interno delle opere d'arte (sensori di temperatura, pressione, umidità...) per poter raccogliere dati utili al monitoraggio della struttura stessa e per eventuali analisi. Infine, si vuole che il sistema sia in grado di avvisare l'utente nel caso in cui ci siano anomalie o superamenti di livelli di soglia riguardanti i dati che vengono raccolti nel tempo, per rendere il monitoraggio dell'opera d'arte ancora più efficace.

1.3 Contenuto del manuale tecnico

Il manuale tecnico è articolato come segue:

- Nel Capitolo 1, viene data una visione di insieme alle motivazioni che hanno portato alla realizzazione di questo progetto;

- Nel Capitolo 2, viene analizzata la natura del problema che viene affrontato in questo progetto;
- Nel Capitolo 3, viene presentata la soluzione pensata ed adottata per risolverle il problema presentato nel capitolo precedente e l'architettura scelta per la realizzazione del sistema in questione;
- Nel Capitolo 4, si passa alla descrizione dettagliata dell'implementazione della soluzione proposta nel capitolo precedente, soffermandosi sull'implementazione di un caso d'uso specifico.

Analisi del problema

2.1 Definizione del problema

Il progetto REMOTE si pone l'obiettivo di rappresentare nella maniera più fedele possibile un'opera d'arte sia dal punto di vista della rappresentazione 3D, che dal punto di vista del modello dati corrispondente. Alla base di tutto, è fondamentale partire da un file *IFC* (*Industry Foundation Classes*) dato che contiene la maggior parte delle informazioni riguardanti l'opera d'arte da trattare, le relazioni tra i vari componenti che vanno a formare l'opera d'arte stessa e anche perché è lo standard attuale per quanto riguarda i *BIM (Building Information Modeling)*. Spostandosi sulla parte riguardante il modello dati da adottare, per la struttura da rappresentare, visto che si vuole avere una rappresentazione completa, è importante avere un meccanismo che permetta di ottenere tutti i dati contenuti all'interno del file *IFC* da cui si deve partire, in maniera tale da non avere una rappresentazione parziale dell'opera d'arte da trattare. Affianco a questo modello dati da memorizzare all'interno di un database, non meno importante è la memorizzazione di tutte le informazioni riguardanti i parametri di monitoraggio di un'opera d'arte, come la temperatura, l'umidità o la pressione all'interno di una struttura, in maniera tale da storicizzare sia i valori raccolti dalla simulazione di alcuni sensori e le relative serie temporali. Tutto questo è necessario per garantire la possibilità di effettuare un monitoraggio della struttura e di alcuni suoi parametri nel corso del tempo ed eventualmente usare questi dati per effettuare analisi sul degrado dei componenti che formano la struttura o sui vari fenomeni che si presentano all'interno di quest'ultima. Inoltre, per rendere ancora più efficace il monitoraggio, il progetto REMOTE ha anche lo scopo di tenere aggiornato l'utente che avvia il monitoraggio dell'opera d'arte da trattare attraverso l'utilizzo di notifiche come email, SMS o altro. Dal punto di vista della visualizzazione, anche l'integrazione di dashboard apposite è sicuramente utile per migliorare l'esperienza dell'utente e per permettere di tenere sotto controllo tutti i dati che vengono raccolti dai vari sensori, in modo tale da avere maggiore contezza di quello che sta succedendo all'interno della struttura. Sempre per la visualizzazione, la rappresentazione 3D del modello da utilizzare è utile e necessaria per permettere all'utente di visualizzare meglio il modello, permettendo di capire esattamente come i vari componenti sono posizionati all'interno della struttura ed eventualmente capire in che modo sono relazionati tra loro attraverso

apposite interrogazioni, oltre a come è collocata nello spazio la struttura stessa.

2.2 Requisiti

2.2.1 Requisiti non funzionali

Di seguito verranno presentati i requisiti non funzionali attribuiti a questo progetto. C'è da precisare che trattandosi di un prototipo, il focus non è posto sul soddisfacimento stretto di questi requisti. Essi sono:

- **Manutenibilità**: Andando ad avere un software modulare e di conseguenza suddividendo le operazioni in aree funzionali è semplice individuare una modifica o correzione da apportare ad una singola porzione di codice. Di conseguenza le singole modifiche hanno l'obiettivo di coinvolgere solo determinate porzioni di codice. Tutto è reso più semplice garantendo anche una corretta documentazione relativa al software;
- **Scalabilità**: Garantendo questo requisito si permette di agevolare l'aggiunta di funzionalità e requisiti anche dopo l'implementazione dell'applicazione senza andare a stravolgere il codice di base;
- **Usabilità**: E' importante che l'interfaccia dell'applicazione sia organizzata in maniera tale da rendere ogni operazione semplice da realizzare potendo guidare l'utente in ogni passo da effettuare;
- **Portabilità**: L'applicazione deve essere realizzata in maniera tale da poter garantire che essa venga eseguita nello stesso modo su dispositivi differenti tra loro;
- **Affidabilità**: L'applicazione deve garantire che lo svolgimento di tutte le sue funzionalità avvenga con continuità;

2.2.2 Requisiti funzionali

Funzionalità principali

Queste funzionalità derivano dalla spiegazione e definizione del problema trattata nella Sezione 2.1 e sono riportate in questa sezione:

- **Mappatura di un file IFC**: l'operazione di mapping di un file *IFC* ha come obiettivo quello di far inserire all'utente, un file *IFC* presente sul suo computer. Successivamente l'utente dovrà attendere il completamento dell'operazione, che per file *IFC* complessi potrebbe richiedere del tempo. Al termine del mapping l'utente, sarà reindirizzato in una pagina che mostrerà tutte le operazioni che può selezionare ed effettuare;
- **Visualizzazione del modello 3D della struttura da trattare**: attraverso questa operazione, l'utente avrà la possibilità di utilizzare il

visualizzatore messo a disposizione dal progetto REMOTE che permette di caricare un modello a disposizione e successivamente navigarlo, potendo osservare i componenti che formano il modello e le relative informazioni legate ai singoli componenti, tramite apposite interrogazioni. Inoltre l'utente ha la possibilità anche di rimuovere un modello inserito ed eventualmente caricarne uno differente;

- **Visualizzazione dei dati immagazzinati tramite dashboard**: con questa operazione si vuole dare la possibilità all'utente di visualizzare i dati immagazzinati, relativi ad alcuni parametri di monitoraggio della struttura, attraverso delle dashboard. Oltre a questo, l'utente in questa parte può avviare manualmente la simulazione dei vari sensori attraverso un pulsante che successivamente avvia anche la visualizzazione delle dashboard;

- **Ricezione di notifiche relative ai dati di monitoraggio**: attraverso questa operazione, il sistema è in grado di inviare notifiche riguardanti i dati che vengono raccolti, attraverso l'invio di avvisi su un canale di comunicazione, come email, SMS, o altri canali scelti dall'utente tramite un modulo accessibile sulla pagina web prima di far iniziare realmente la simulazione, nel momento in cui vengono superate determinate soglie preimpostate per i dati raccolti o eventualmente configurate dall'utente.

Funzionalità accessorie

In questa sezione vengono esposte le funzionalità di supporto necessarie all'applicazione.

- **Login**: attraverso l'operazione di autenticazione, l'utente può accedere, attraverso credenziali di amministrazione, svolgere tutte le operazioni consentite dall'applicazione.

Progettazione

3.1 Casi d'uso

Attraverso i casi d'uso verranno presentati gli scenari che riguardano l'interazione tra l'utente e il sistema.

Login:
Attore principale: Utente
Motivazione d'uso: Autenticarsi per effettuare le operazioni messe a disposizione dal sistema
Pre-condizione: Il sistema contiene le credenziali di accesso
Post-condizione: Il sistema riconosce l'utente
Flusso principale:

1. L'utente inserisce nome utente e password
2. Il sistema riconosce l'utente
3. Il sistema mostra l'area dedicata alle operazioni dell'utente

Estensione: L'utente è esistente, ma la password inserita è errata

2a. Il sistema mostra un messaggio di errore

Il flusso riprende da 1

Estensione: L'utente è esistente, ma il nome utente inserito è errato

2a. Il sistema mostra un messaggio di errore

Il flusso riprende da 1

Mapping file IFC:
Attore principale: Utente
Motivazione d'uso: Avviare il mapping del file IFC e ottenere il modello dati corrispodente nel database
Pre-condizione: L'utente ha effettuato il login
Post-condizione: Il sistema ha mappato correttamente il file IFC nel database
Flusso principale:

1. Il sistema mostra la pagina per caricare un file IFC

2. L'utente carica il file IFC
3. L'utente avvia la procedura di mapping del file
4. Il sistema al termine del mapping mostra la pagina contente le operazioni che può svolgere l'utente

Estensione: L'utente inserisce un file IFC già mappato

Il flusso riprende da 4

Estensione: L'utente inserisce un file che non ha l'estensione .ifc

4a. Il sistema mostra un messaggio di errore

Il flusso riprende da 1

Visualizzazione modello 3D:
Attore principale: Utente
Motivazione d'uso: Effettuare la visualizzazione 3D del modello a partire da un file IFC
Pre-condizione: L'utente ha effettuato il login
Post-condizione: Il sistema mostra il modello 3D della struttura
Flusso principale:

1. Il sistema mostra la pagina per utilizzare il visualizzatore
2. L'utente carica il file IFC attraverso il pulsante che permette il caricamento
3. Il sistema carica e mostra il modello

Estensione: L'utente rimuove il modello inserito dal sistema

4. L'utente preme il pulsante che mostra i modelli caricati
5. L'utente seleziona un modello e lo elimina

Il flusso riprende da 1

Estensione: L'utente visualizza le proprietà del modello

4. L'utente clicca su un frammento del modello 3D che viene evidenziato
5. L'utente preme il pulsante che visualizza le proprietà

Il flusso riprende da 1

Visualizzazione dei dati immagazzinati tramite dashboard:
Attore principale: Utente
Motivazione d'uso: Effettuare la visualizzazione delle dashboard riguardanti i dati di temperatura generati
Pre-condizione: L'utente ha effettuato il login
Post-condizione: Il sistema mostra le dashboard in tempo reale
Flusso principale:

1. Il sistema mostra la pagina per utilizzare le dashboard
2. Il sistema mostra il form per inserire la mail da notificare, la soglia di temperatura e il numero di dati da generare.
3. L'utente fa partire la simulazione del sensore selezionato attraverso il pulsante messo a disposizione
4. Il sistema mostra le dashboard che vengono aggiornate in tempo reale

Estensione: L'utente ritorna alla pagina delle operazioni

2. L'utente preme il pulsante per tornare alla pagina delle operazioni

Estensione: L'utente riceve una notifica dal sistema

4. Superata una certa soglia di temperatura impostata, il sistema invia una notifica all'utente

   Il flusso riprende da 3

Estensione: L'utente inserisce una mail con formato errato

3a. Il sistema mostra un messaggio di errore

   Il flusso riprende da 2

Di seguito verranno illustrati dei diagrammi UML riguardanti alcuni casi d'uso esposti sopra nella Figura 3.1 e Figura 3.2. I casi d'uso sono una tecnica utilizzata per individuare i requisiti funzionali di un sistema e si basano sulla descrizione delle interazioni tra gli utenti e il sistema. Un caso d'uso è un insieme di scenari che hanno in comune lo scopo finale dell'utente. Di preciso, ogni utente è un attore che indica il ruolo interpretato da questo in relazione al sistema. Proprio per questa ultima frase in realtà, sarebbe più appropriato anche parlare di ruolo e non di attore [1].

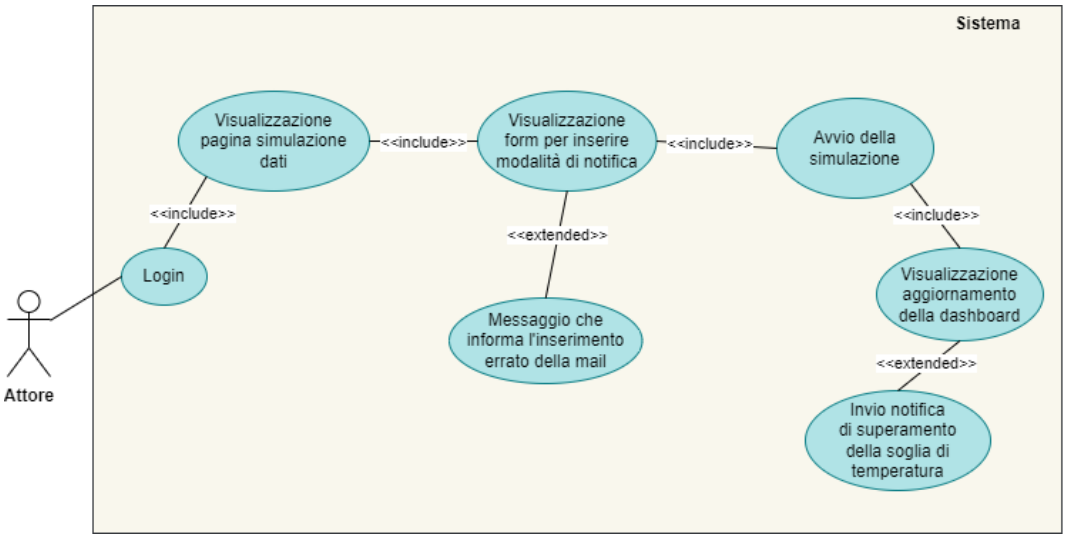

Figure 3.1: Caso d'uso per mappatura del file

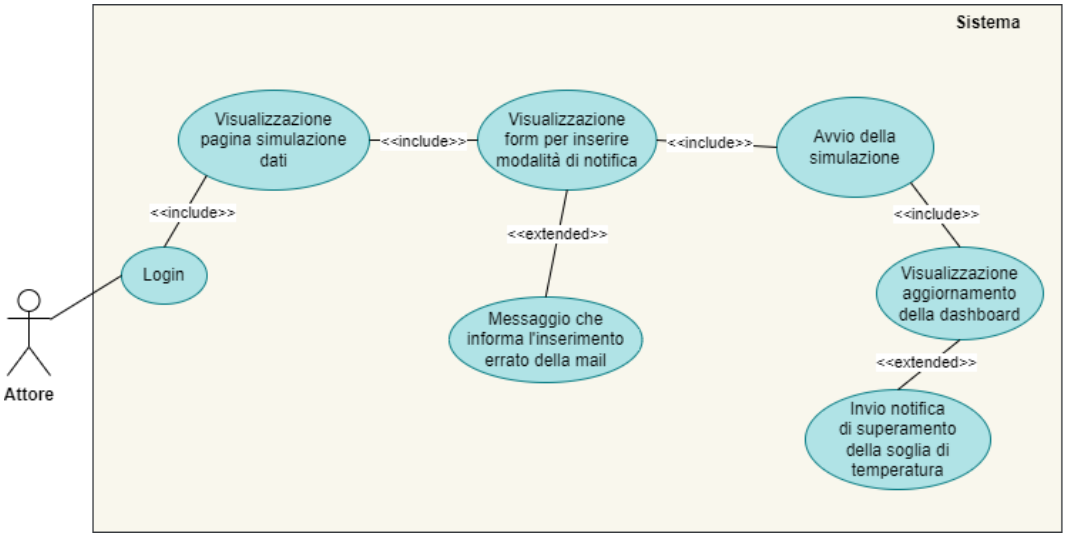

Figure 3.2: Caso d'uso per monitoraggio dati

3.2 Sviluppo e realizzazione dell'IFC-graph

Tutta questa sezione e le sue relative sotto-sezioni fanno riferimento a "*IFC-graph for facilitating building information access and query*" [2]. In questa parte verranno spiegate le scelte alla base dell'algoritmo di mapping utilizzato in questo progetto.
A causa della difficoltà di accesso e di interrogazione dei dati *IFC* attraverso applicazioni esterne all'*AEC* (Architecture, Engineering and Construction), sicuramente le tecnologie messe a disposizione dai database permettono di ottenere in maniera semplice, l'accesso alle informazioni attraverso le API messe a disposizione insieme ai linguaggi di interrogazione. Tra i vari database relazionali e NoSQL, di certo i database a grafo sono quelli che rappresentano meglio le relazioni tra i dati IFC, adattandosi nel miglior modo. Oltre a questo, tra le due tecnologie dei database a grafo, *RDF* (Resource Description Framework) e *LPG* (Labeled Property Graph), quella *LPG* è la più adatta. Le motivazioni sono sostanzialmente due. La prima è incentrata sulla necessità di facilitare l'accesso e l'interrogazione delle informazioni delle strutture. La seconda fa leva sul fatto che un'interrogazione, per ottenere i dati in maniera efficiente, richiede un modello dati a grafo per rappresentare i dati del modello *IFC* originale. Su questi due aspetti, la tecnologia *LPG* può funzionare meglio di quella *RDF*, dato che *RDF* scompone i vari componenti in tante unità di base. L'approccio adottato è mostrato in Figura 3.3. Esso è sostanzialmente composto da 3 step:

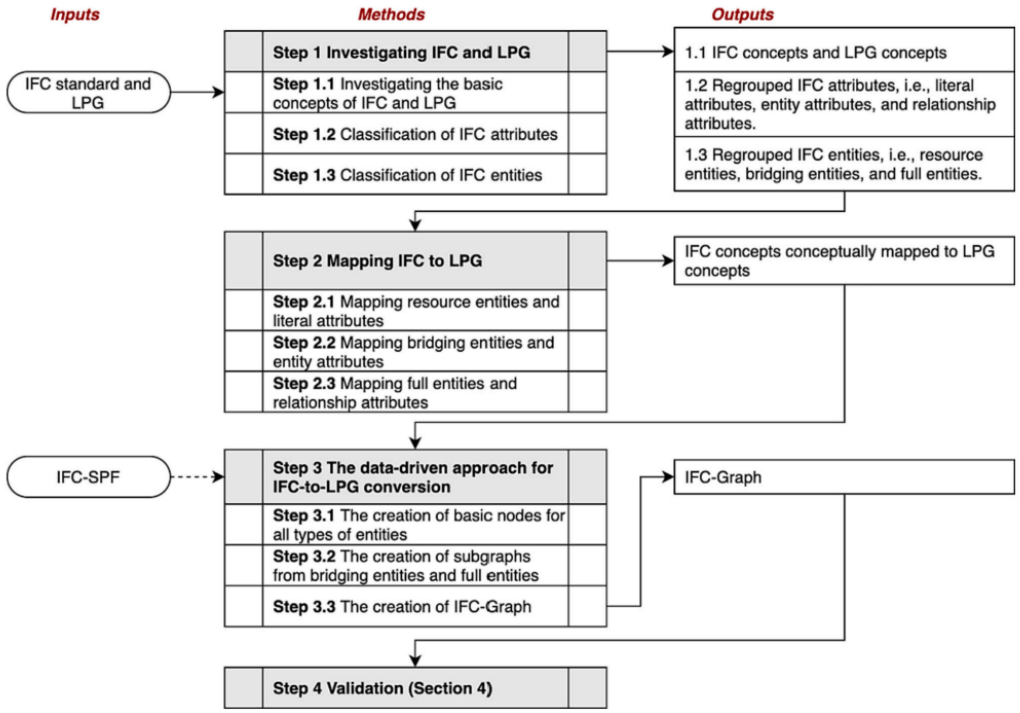

Figure 3.3: Diagramma di approccio al mapping [2]

1. Investigazione del file *IFC* e dell'*LPG* per capire la struttura di entrambi;
2. Effettuare il mapping concettuale dei concetti *IFC* in concetti *LPG*;
3. Sviluppare gli algoritmi per implementare la conversione *IFC-LPG*.

Con questo approccio, il modello determina le informazioni che devono essere convertite a l'*LPG* generato sarà la rappresentazione completa dei dati *IFC*, che si troveranno all'interno dell'IFC-graph.

3.3 Analisi dell'IFC e dell'LPG

IFC è uno standard internazionale per lo scambio di informazioni sulle strutture con il dominio *AEC*. Esso è formato da due parti, l'**IFC schema** e l'**IFC data**. Lo schema IFC definisce le entità/classi e le loro relative relazioni, mentre i dati IFC sono una descrizione specifica delle informazioni di una struttura utilizzando le entità definite dall'IFC schema. Il Listato 3.1 mostra un esempio. Nella sezione DATA, ogni linea indica un'istanza di una certa entità IFC, al quale si può fare riferimento attraverso il suo identificatore unico (#ID). Questo meccanismo di riferimento, permette di connettere le varie istanze tra loro ed eventualmente creare anche una rete complessa.

Listato

3.1

Listing 3.1: Esempio di file IFC

```
---
ISO-10303-21;
HEADER;
FILE_DESCRIPTION(('ViewDefinition [CoordinationView_V2.0]'),'2;1');
FILE_NAME('0001','2024-02-09T14:09:27',(''),(''),'ODA IFC SDK 22.8','22.1.30.34
- Exporter 22.1.30.34 - Alternate UI 22.1.30.34','');
FILE_SCHEMA(('IFC2X3'));
ENDSEC;

DATA;
#1=IFCORGANIZATION($,'Autodesk Revit 2022 (ITA)',$,$,$);
#2=IFCAPPLICATION(#1,'2022','Autodesk Revit 2022 (ITA)','Revit');
#3=IFCCARTESIANPOINT((0.,0.,0.));
...
...
#922=IFCLOCALPLACEMENT(#917,#921);
ENDSEC;
END-ISO-10303-21;
---
```

Un *LPG* consiste in un insieme di *nodi* (o vertici) e uno di relazioni (o *edges*). Ogni nodo rappresenta un oggetto, astratto o concreto. Ovviamente ogni nodo può contenere più coppie chiave-valore per le varie informazioni contenute nell'IFC. Le relazioni (edges) nell'LPG sono direzionate e sono usate per collegare i nodi, le quali possono avere un nome e più coppie chiave-valore per memorizzare informazioni. Le strutture dell'IFC data sono adatte ad essere rappresentate con quelle dell'LPG. Infatti, le istanze dell'IFC-SPF possono essere rappresentate usando i nodi, mentre gli attributi che collegano le varie istanze possono essere rappresentate attraverso le relazioni dell'LPG. Questo tipo di mapping strutturale è abbastanza semplice, cosa che la mappatura concettuale non è, dato che ci sono delle differenze semantiche legate a dei concetti chiave come "relazione", "attributo" e "proprietà".

3.3.1 Riclassificazione degli attributi IFC

In IFC, un attributo è "l'unità di informazione all'interno di un'entità, definita da un particolare tipo o riferita ad una particolare entità". Ci sono tre tipi di attributi definiti nell'IFC, includendo **attributi diretti**, **inversi** e **derivati** che sono classificati relativamente al tipo dell'attributo. Questo tipo di classificazione non aiuta l'approccio model-driven. Pertanto è necessario riclassificare gli attributi in tre gruppi riferiti ai valori assunti dai vari tipi di dato includendo gli *attributi letterali*, gli *attributi dell'entità* e quelli della *relazione*. Gli *attributi letterali* sono quegli attributi diretti e derivati i quali data type, ad esempio, possono essere text, number e bool come gli attributi "GlobalId" e "Name" dell'entità IfcRoot. Gli *attributi dell'entità* sono derivati dagli attributi diretti i quali valori sono alte istanze o gruppi di istanze, ad esempio gli attributi "ObjectPlacement" e "Representation" dell'entità IfcProduct. Gli *attributi di relazione* sono gli attributi inversi, come "IsDecomposedBy" nell'entità IfcObjectDefinition. Essi fanno riferimento alle istanze delle relazioni. Ogni entità IFC ha attributi letterali e potrebbe avere attributi di entità e/o attributi di relazione. Gli attributi letterali sono espliciti nell'IFC-SPF, mentre gli attributi di relazione sono impliciti.

3.3.2 Classificazione delle entità IFC

Per facilitare il mapping da IFC a grafo, le entità IFC devono essere raggruppate in tre tipi per essere in linea con i tipi di attributi discussi prima. Questi gruppi includono:

1. Le entità con soli attributi letterali come IfcUnit e IfcValue. Queste entità di solito si ottengono dal resource layer dell'architettura dello schema IFC e sono usate per memorizzare i valori degli attributi letterali. Queste entità non hanno un identificativo univoco e possono essere chiamate **resource entities**;

2. Le entità che hanno soltanto gli attributi letterali e di entità. Esse sono, nell'IFC, le IfcRelationship (relazioni) e sono usate per collegare gli oggetti. Possono essere chiamate **bridging entities**;

3. Le entità che hanno tutti e tre gli attributi. Queste derivano da oggetti (IfcObjectDefinition) e da proprietà (IfcPropertyDefinition). Queste entità possono essere chiamate **full entities**.

Questi tipi di entità sono mostrate anche in Figura 3.4. Questa classificazione permette di semplificare il mapping dei concetti IFC in quelli LPG e di conseguenza anche l'implementazione della conversione model-driven da IFC a LPG.

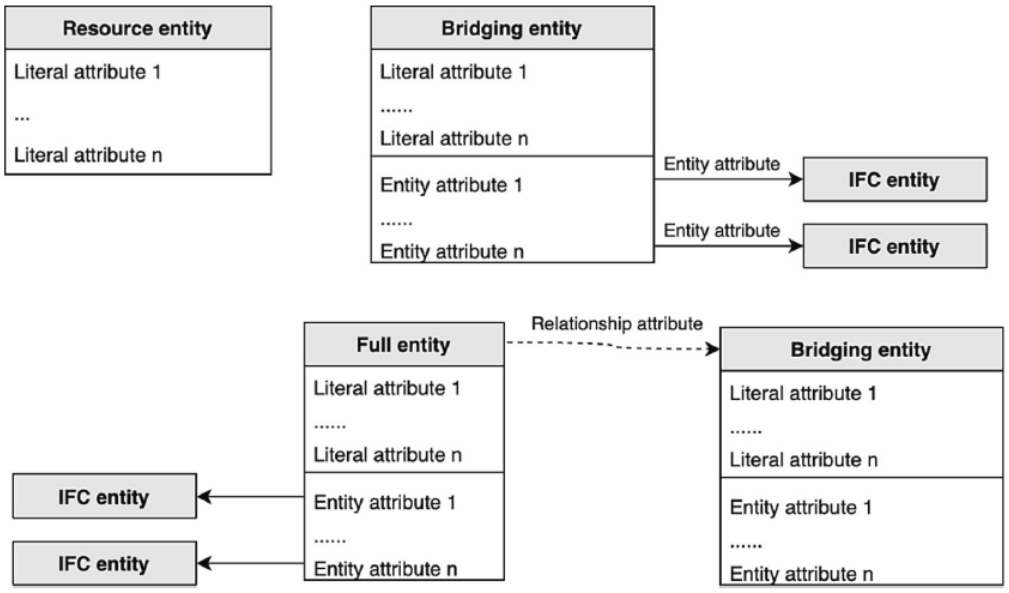

Figure 3.4: Tipi di entità nel file IFC [2]

3.4 Mapping dell'IFC in LPG

3.4.1 Mapping delle resource entities e degli attributi letterali

Gli attributi letterali sono mappati come proprietà dei nodi, con l'obiettivo di ridurre la dimensione del grafo, che è vitale per ottenere informazioni tramite query basate su grafo. La Figura 3.5 mostra un esempio di questo mapping sull'entità IfcPerson. Il risultato di questo mapping sono dei nodi con esclusivamente delle proprietà.

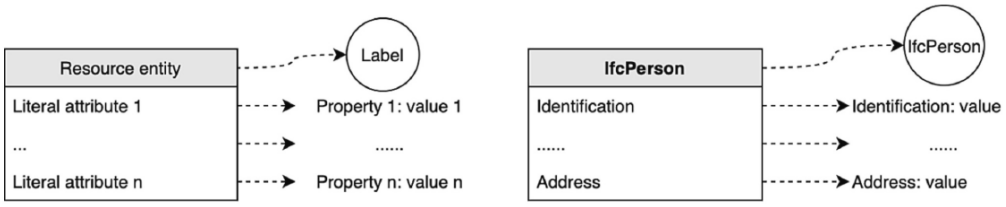

Figure 3.5: Mappatura delle entità [2]

3.4.2 Mapping delle bridging entities e degli attributi di entità

Oltre agli attributi letterali, le bridging entities hanno in aggiunta gli attributi di entità. Questi attributi, sono mappati come relazioni nel grafo. Per una struttura formata dalla coppia delle bridging entities e le loro entità referenziate, la bridging entity iniziale diventa l'entità primaria, mentre le altre entità referenziate diventano sotto-entità. I corrispondenti nodi creati saranno quindi nodi primari e sotto-nodi, rispettivamente. Dopo il mapping, le relazioni tra i nodi primari e i sotto-nodi, sono dirette dai nodi primari ai sotto nodi, come mostrato in Figura 3.6. A differenza del mapping delle resource entities, il quale risultato dà come risultato un nodo di base, il risultato di questo mapping è un grafo semplice formato dai nodi primari e dai sotto-nodi, con relative relazioni tra loro. Questo grafo, è una parte del grafo finale, pertanto si può fare riferimento ad esso come sotto-grafo.

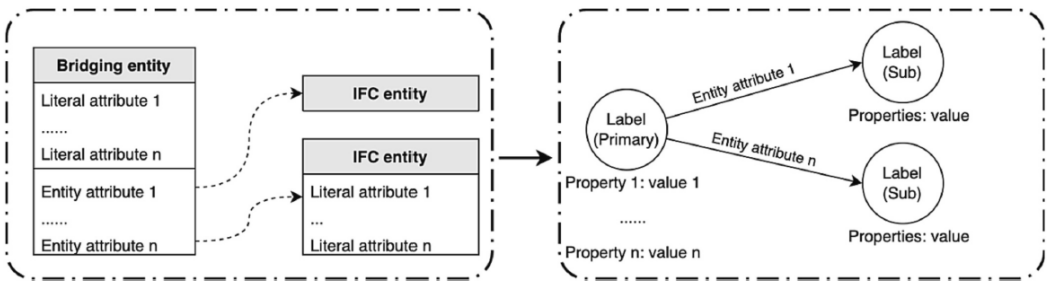

Figure 3.6: Mappatura delle entità e degli attributi di collegamento [2]

3.4.3 Mapping delle full entities e degli attributi di relazione

Rispetto alle bridging entities, le full entities hanno in aggiunta gli attributi di relazione. Gli attributi di relazione sono mappati nelle relazioni del grafo con i loro nomi, andando ad indicare i tipi delle relazioni. Il risultato di questo mapping è un sotto-grafo, come mostrato in Figura 3.7. Dopo il mapping, le entità IFC, indipendentemente dai loro tipi, risultato essere nello stesso tipo di nodo, cioè nodi che hanno esclusivamente delle proprietà, ma con differenti tipi di relazioni. Le resource entities sono dei nodi di base con delle relazioni in entrata, mentre le bridging entities e le full entities sono sempre nodi di base ma con relazioni sia in entrata che in uscita. La differenza tra le bridging entities e le full entitites in termini di relazione è che le bridging entities non posso direttamente fare riferimento ad altre bridging entities, mentre le full entitites posso direttamente fare riferimento a qualsiasi tipo di entità.

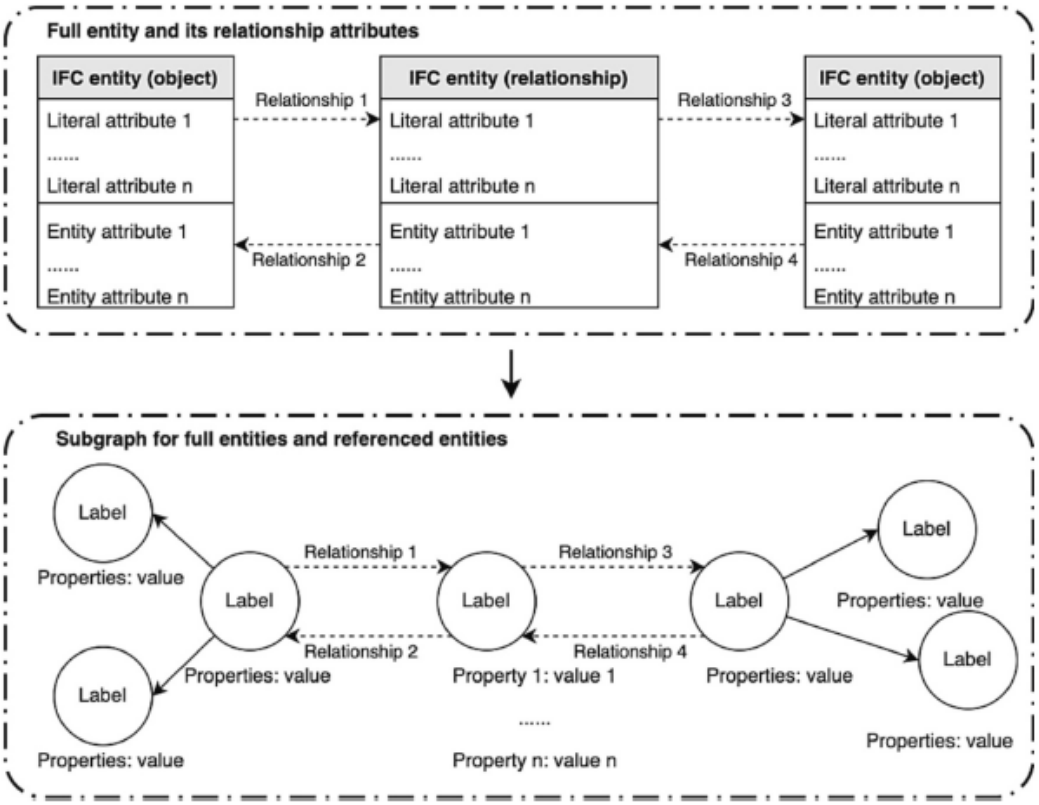

Figure 3.7: Mapping delle full entities e degli attributi di relazione nel grafo [2]

3.5 L'approccio model-driven per la conversione IFC-LPG

L'approccio model-driven alla base della conversione da IFC ad LPG ha come punto di partenza l'IFC-SPF, da cui si ottengono le informazioni da convertire,

Figura 3.8. Tutti i contenuti di un IFC-SPF vengono convertiti. Questa conversione, generalmente coinvolge due step: l'estrazione delle informazioni IFC e la generazione del grafo. L'estrazione delle informazioni IFC viene realizzata sfruttando IfcOpenShell, e tutte le istanze, come i loro attributi vengono analizzati per garantire una conversione completa. Il grafo successivamente viene generato in tre step:

1. La creazione dei nodi per istanze individuali;
2. La creazione di sotto-grafi per le bridging entities e le full entities;
3. La combinazione dei vari sotto-grafi per formare il grafo finale

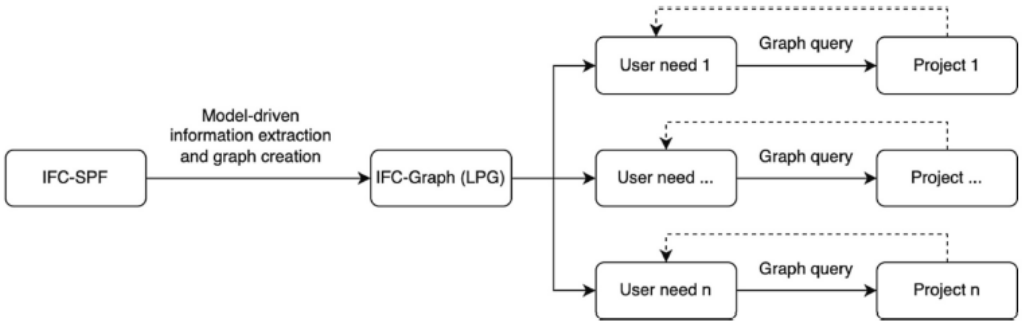

Figure 3.8: Approccio model-driven per la conversione IFC-LPG [2]

3.5.1 La creazione dei nodi di base per tutti i tipi di entità

La creazione dei nodi di base è l'operazione più importante e ha due aspetti delicati. In primo luogo, oltre agli attributi letterali di base, la gerarchia delle entità IFC dovrebbe essere preservata per garantire un recupero delle informazioni, attraverso query, efficiente. In secondo luogo, le resource entities che non hanno un identificativo unico globale (id) devono essere gestite correttamente:

- Conservazione della gerarchia di entità: una gerarchia di entità ben conservata è utile per le query su grafo utilizzate per recuperare le informazioni delle strutture. Per esempio, per recuperare tutti i componenti di una struttura dal grafo, la classe astratta "IfcBuildingElement" può essere usata, senza prendere tutte le classi individuali più importanti, come IfcWall, IfcWindow e IfcDoor.
- Elaborazione delle resource entities: le resource entities, come IfcLabel, IfcDate e IfcUnit, non hanno un id unico. Pertanto, quando vengono processate le istanze di queste entità, un identificativo arbitrario basato su testo viene assegnato per due scopi: identificare univocamente le resource entities, per non creare nodi duplicati, e per distinguerli dalle istanze normali che hanno un id numerico.

3.5.2 La creazione dei sotto-grafi per bridging entities e full entities

Basandosi sugli step precedenti, un sotto-grafo, per ogni istanza delle bridging entities e delle full entities, può essere creato. Per ogni istanza, la procedura usata per generare il sotto-grafo è la seguente:

1. Creare un nodo di base per l'istanza, che è il nodo primario;

2. Analizzare i sui attributi di entità. Per ogni attributo di entità trovato, si crea un nodo di base per l'istanza a cui si fa riferimento, che sarà il sotto-nodo, e si stabilisce la relazione tra loro, che andrà dal nodo primario al sotto-nodo.

3. Analizzare gli attributi di relazione. Per ogni attributo di relazione trovato, si crea un nodo di base per l'istanza della relazione a cui si fa riferimento, come la relazione tra loro. Da notare che l'istanza di IfcOwnerHistory è riferita solo a quella di IfcProject per migliorare l'efficacia delle interrogazioni, mentre nell'IFC fa riferimento a più istanze di IfcRoot. Quando si processano questi due tipi di entità IFC, i risultati sono dei sotto-grafi, che possono essere descritti da un modello di grafo di tipo primary-to-sub, Figura 3.9. In questo grafo atomico, il nodo primario è per l'entità primaria che viene processata, mentre i sotto-nodi (nodi di riferimento) sono le entità IFC referenziate. Per i nodi primari ci sono solo relazioni in uscita.

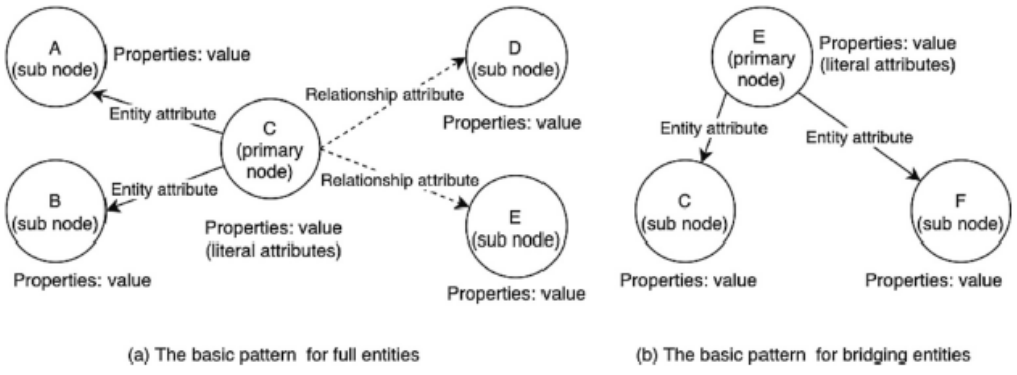

Figure 3.9: Modello di grafo primary-to-sub per (a) le full entites e (b) bridging entities [2]

3.6 Creazione dell'IFC-graph

Il grafo IFC finale può essere creato andando a fare il merging graduale dei nuovi sotto-grafi con il grafo esistente. Figura 3.10 mostra il merge dei due grafi mostrati in Figura 3.9 come rappresentazione di esempio del processo.

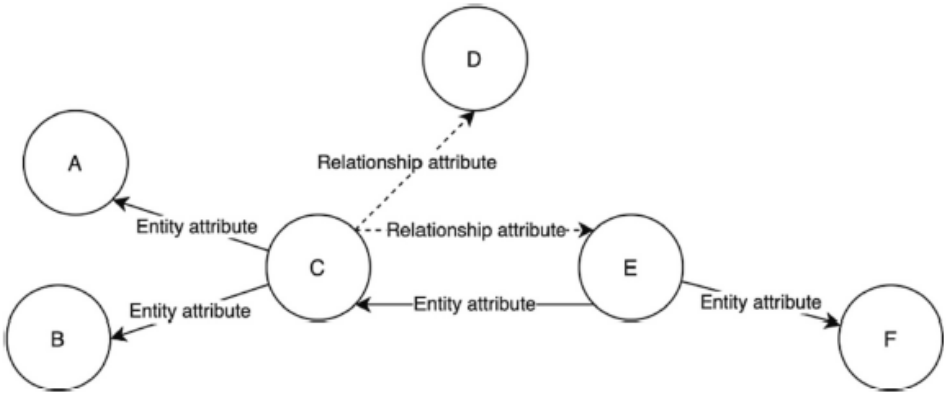

Figure 3.10: Grafo generato usando l'approccio model-driven [2]

Nell'IFC-graph le relazione sono direzionate ed unidirezionali e una relazione bidirezionale tra due nodi viene stabilita in due step:

1. La direzione dal nodo C al nodo E ad esempio viene stabilita quando C è entità primaria;

2. La direzione da E ad C viene stabilita quando E è entità primaria.

È importante sottolineare che la definizione dei tre tipi di entità viene fatto esclusivamente per semplificare il processo di conversione rendendo anche più semplice l'implementazione dell'algoritmo. Infatti queste entità non sono definite nei dati originali di IFC.

3.7 Diagramma di sequenza

In questa sezione viene mostrato il diagramma di sequenza, in Figura 3.11, che tratta il caso d'uso della mappatura di un file IFC nel database a grafo scelto per questo progetto. I diagrammi di sequenza rientrano tra i diagrammi di interazione che descrivono la collaborazione di un gruppo di oggetti che devono implementare collettivamente un certo comportamento. Nello specifico, i diagrammi di sequenza trattano la documentazione del comportamento di un singolo scenario [1].

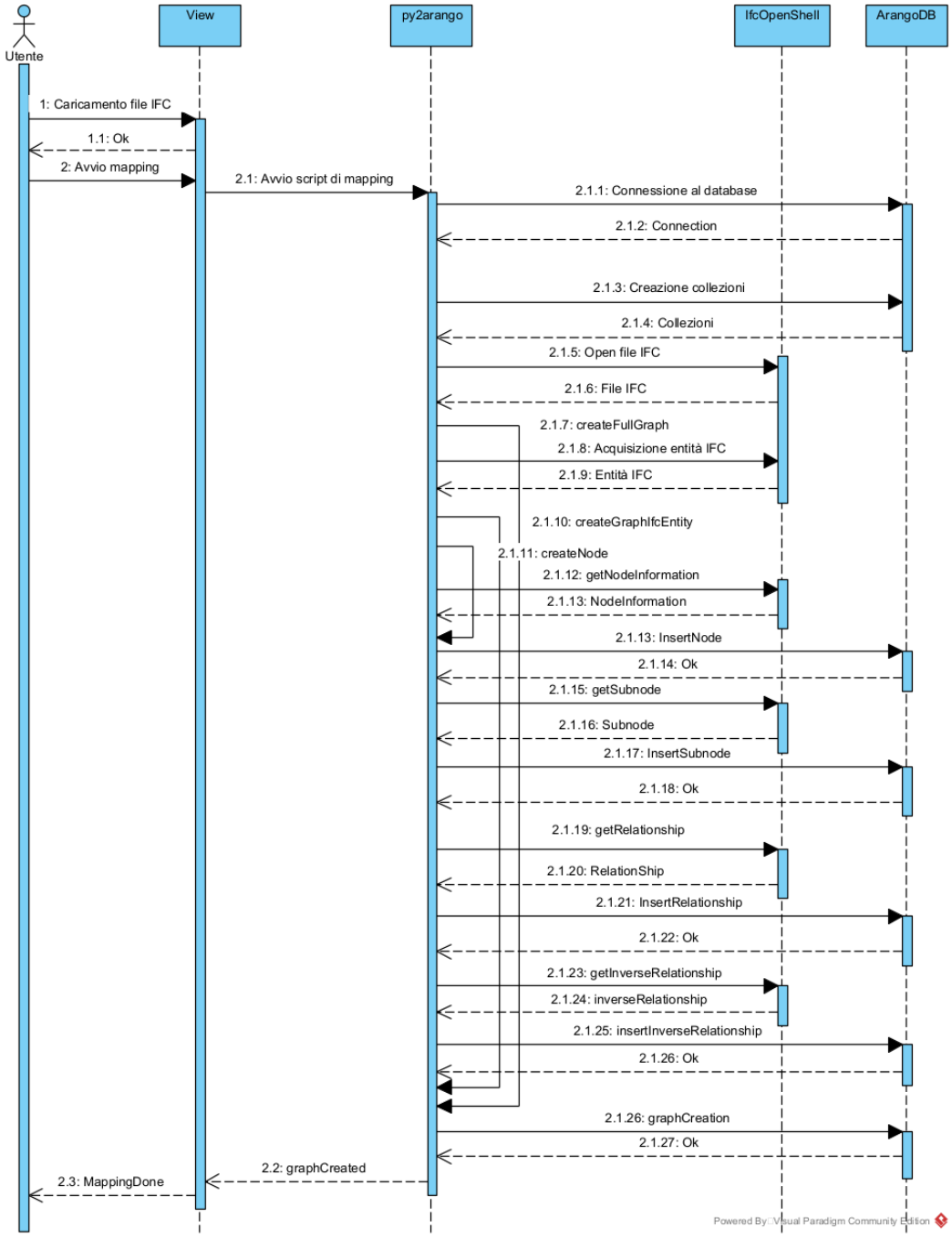

Figure 3.11: Diagramma di sequenza

3.8 Wireframes

Oltre alla realizzazione dei vari diagrammi UML, sono stati realizzati anche dei wireframes per avere una bozza dell'interfaccia grafica che successivamente sarà implementata nel Capitolo 4. Qui vengono mostrati soltanto alcuni wireframes come esempio.

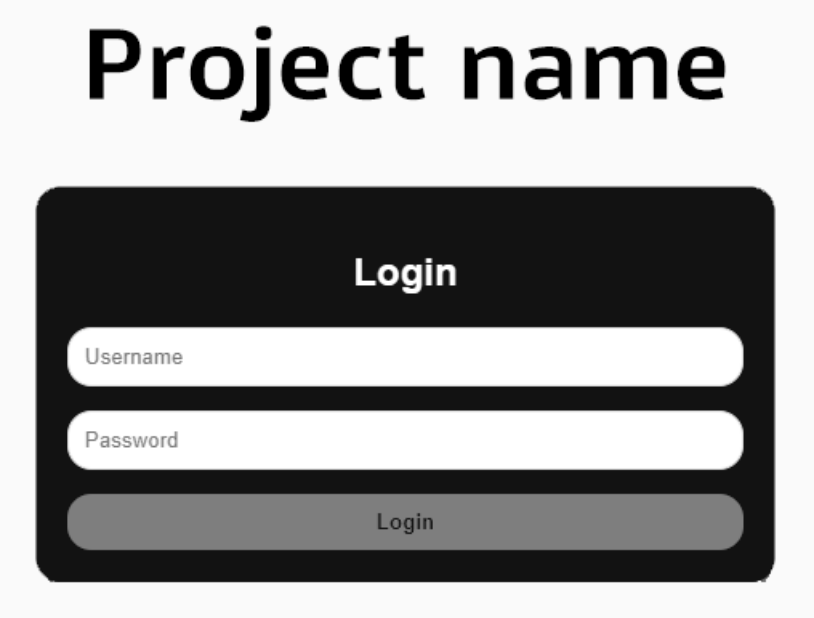

Figure 3.12: Wireframe per la pagina di login

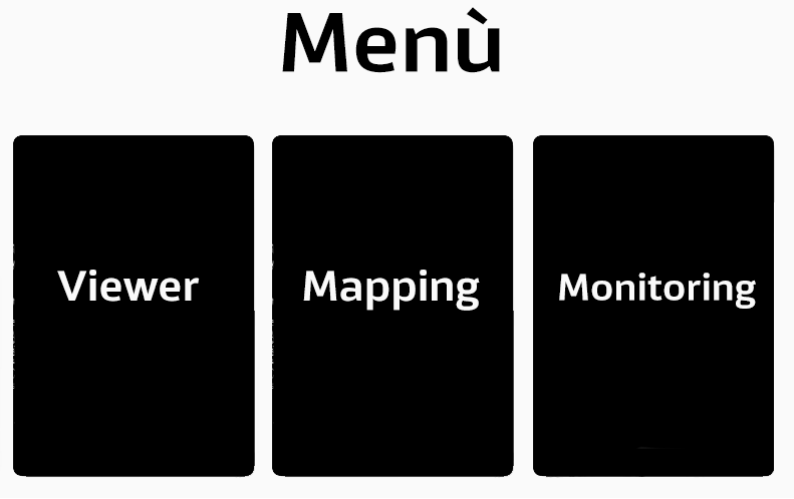

Figure 3.13: Wireframe per la pagina del menù

Figure 3.14: Wireframe per la pagina di mapping

3.9 Soluzione proposta

Dopo aver presentato le problematiche, che sono state esposte nella Sezione 2.1, di seguito viene presentata la soluzione pensata per questo progetto. Per soddisfare le esigenze emerse nell'analisi dei requisiti descritta nella Sezione 2.2, si propone di realizzare una web-application, che permetta di realizzare il mapping dei file IFC, la visualizzazione del modello 3D derivante e simulare il comportamento di un sensore di temperatura, con relativa visualizzazione delle dashboard attraverso l'applicazione stessa. Durante lo svolgimento di queste operazioni, i dispositivi che vengono utilizzati per usare l'applicazione devono essere dotati di connessione Internet, per poterle concludere con successo e memorizzare tutti i dati raccolti sugli appositi database utili, rispettivamente per immagazzinare il modello mappato del file IFC e i dati relativi al sensore di temperatura simulato nell'applicazione REMOTE. L'utente prima di poter accedere a tutte le operazioni disponibili, deve effettuare l'autenticazione attraverso le credenziali di amministrazione memorizzate nell'applicazione. Dopo l'autenticazione, l'utente può scegliere di effettuare il mapping di un file IFC selezionandolo correttamente dal suo dispositivo, pertanto il file deve essere già presente sul dispositivo al momento del caricamento. Una volta effettuato il mapping, tra le altre operazioni l'utente può decidere di selezionare il visualizzatore 3D. Questa operazione permette all'utente di interagire con il visualizzatore 3D dell'applicazione REMOTE che permette di caricare un file IFC. Una volta caricato, l'utente può decidere di rimuovere il modello caricato attraverso l'apposito pulsante oppure selezionare dei componenti del modello e osservare le caratteristiche di questi come il tipo di materiale, lo stile, le misure, il nome del componente, ecc... e le relazioni con gli altri componenti attraverso apposite interrogazioni. Oltre a queste due operazioni, l'utente può anche effettuare delle misurazioni sulle distanze tra i componenti o un componente

stesso. Infine, come altra operazione l'utente può scegliere quella dedicata al monitoraggio, dove l'applicazione mostra le dashboard dedicate alla visualizzazione dei dati raccolti. Inoltre, l'utente può decidere quando far partire la simulazione e selezionare il tipo di notifica da ricevere (es. email) impostando una certa soglia, basata sui dati generati, oltre il quale ricevere la notifica e il numero di dati da generare.

3.10 Architettura del sistema

3.10.1 Introduzione

In questa sezione, verrà presentata l'architettura adottata per la realizzazione della soluzione proposta nella Sezione 3.9.
L'architettura adottata per questo progetto è basata su microservizi la quale, scompone l'applicazione in una serie di componenti modulari e indipendenti, noti come microservizi. Ogni microservizio svolge una funzione specifica e comunica con gli altri tramite interfacce ben definite, come API HTTP o altro. Uno dei principali vantaggi di questa architettura è la scalabilità.
Grazie alla modularità dei microservizi, è possibile scalare orizzontalmente solo i componenti che necessitano di maggiori risorse, senza dover scalare l'intera applicazione. Inoltre, l'approccio a microservizi favorisce l'agilità nello sviluppo. Poiché ogni microservizio è un'unità autonoma, il lavoro può essere condotto in maniera indipendente su diverse parti dell'applicazione.
Ciò consente un'implementazione più rapida di nuove funzionalità e una maggiore flessibilità nel rispondere ai cambiamenti dei requisiti. La resilienza e l'affidabilità sono ulteriori vantaggi di questa architettura. Poiché i microservizi sono indipendenti l'uno dall'altro, un guasto in uno di essi non influisce sugli altri. Questo aumenta la resilienza complessiva del sistema e riduce il rischio di un'interruzione dell'intera applicazione.
Un'altra caratteristica importante è l'utilizzo delle tecnologie più adatte al compito. Ogni microservizio può essere implementato utilizzando la tecnologia più adatta alle esigenze specifiche del servizio. La modularità dei microservizi semplifica anche la sostituzione e l'aggiornamento dei componenti. Poiché ogni microservizio è separato dagli altri, è possibile aggiornare la versione di uno di questi senza dover aggiornare l'intera applicazione. Questo riduce il rischio di interruzioni e semplifica la gestione delle modifiche.
Infine, l'architettura basata su microservizi consente un migliore monitoraggio e gestione delle prestazioni. Con microservizi separati per ogni funzionalità, è più facile monitorare e gestire le prestazioni e la disponibilità di ciascun componente.
Ovviamente non sono da trascurare gli aspetti riguardanti la sicurezza, la complessità e l'overhead di comunicazioni tra le varie componenti dell'architettura che sono fondamentali per garantire il corretto funzionamento dell'applicazione e per non intaccare le prestazioni di questa. Di seguito viene data una descrizione delle componenti che formano l'architettura adottata.

3.10.2 ArangoDB

ArangoDB è un sistema di gestione di database multi-modello, open-source e NoSQL. Questo significa che può gestire diversi tipi di dati utilizzando vari modelli, tra cui il modello a documenti, il modello a grafo e il modello a chiave-valore. Con il suo approccio multi-modello, ArangoDB consente di gestire dati strutturati, semi-strutturati e non strutturati all'interno di un singolo database. I dati possono essere memorizzati in formato **JSON** attraverso il modello a documenti, mentre le query vengono eseguite utilizzando il linguaggio di interrogazione **AQL** (ArangoDB Query Language), il quale consente di eseguire interrogazioni, coinvolgendo dati provenienti da diversi modelli di dati. Inoltre, ArangoDB supporta **GeoJSON** e consente di eseguire query geo-spaziali. Questa funzionalità può risultare particolarmente utile per interrogare il database riguardo alla localizzazione dei sensori o alla posizione della struttura nello spazio, ad esempio. Infine, per garantire una maggiore scalabilità e disponibilità, ArangoDB può essere distribuito su più nodi, consentendo così una scalabilità orizzontale e un'alta affidabilità del sistema. Pertanto, ArangoDB è stato scelto per sfruttare le sue caratteristiche di database a grafo per immagazzinare il risultato ottenuto dalla procedura di mapping del file IFC e per mappare anche le relazioni tra i sensori ed eventualmente i componenti del modello mappato con cui vengono messi in relazione.

3.10.3 Flask

Flask è un framework web leggero e flessibile per **Python** che fornisce un'infrastruttura semplice ma potente per lo sviluppo di applicazioni web.
È progettato per essere leggero e senza vincoli, offrendo solo le funzionalità essenziali senza aggiungere complessità superflua. Questa caratteristica lo rende flessibile, consentendo di utilizzare le librerie e gli strumenti che utili a gestire funzionalità come l'accesso al database, la gestione delle sessioni utente e l'autenticazione. Una delle caratteristiche distintive di Flask è la sua estensibilità: segue il principio "plug and play", permettendo di estendere le funzionalità di base del framework aggiungendo estensioni o librerie di terze parti. Questo consente di adattare Flask alle esigenze specifiche di questo progetto.
Flask offre un sistema di routing flessibile che consente di associare funzioni Python a specifici percorsi URL, gestendo le richieste HTTP in modo efficiente e fornendo risposte dinamiche basate su input dell'utente. Supporta inoltre l'utilizzo di template per la generazione dinamica di contenuti HTML, migliorando la manutenibilità del codice separando la logica di presentazione dalla logica di business dell'applicazione. Dal punto di vista della sicurezza, Flask fornisce strumenti per gestire in modo sicuro aspetti critici come la gestione delle sessioni utente, la protezione da attacchi **CSRF** (Cross-Site Request Forgery) e **XSS** (Cross-Site Scripting), e l'autenticazione degli utenti.
All'interno del progetto REMOTE, Flask è stato scelto per implementare tutta la logica di Business del progetto e gestire l'implementazione delle API REST implementate.

3.10.4 Node.js e three.js

Node.js è un ambiente di runtime **JavaScript** open source che consente l'esecuzione di codice JavaScript e TypeScript lato server. A differenza di JavaScript, che è comunemente utilizzato nei browser per creare interattività nelle pagine web, Node.js consente di eseguire JavaScript lato server, gestendo operazioni di I/O in modo asincrono. E infine, **Three.js** è una libreria **JavaScript** open source progettata per semplificare la creazione di grafica **3D** interattiva all'interno di pagine web. Con Three.js, si possono creare scene 3D, aggiungere oggetti, applicare materiali, gestire luci e creare effetti visivi in modo relativamente semplice utilizzando il linguaggio di programmazione JavaScript. Di conseguenza Node.js e three.js sono stati scelti per realizzare la rappresentazione e il rendering 3D dei modelli caricati dall'utente.

3.10.5 Orion-LD

Orion-LD è un context broker e un componente di base di CEF per la gestione dei dati di contesto che supporta sia le API NGSI-LD che le API NGSI-v2 e permette di gestire l'intero ciclo di vita delle informazioni di contesto, inclusi gli aggiornamenti, le interrogazioni, le registrazioni e le sottoscrizioni. . Attualmente è una derivazione dell'originale Orion Context Broker che estende il supporto per aggiungere NGSI-LD e concetti legati ai linked-data. Orion-LD segue la specifica ETSI per NGSI-LD ed è stato testato per essere un broker NGSI-LD stabile e veloce con una stretta conformità alla versione 1.3.1 della specifica dell'API NGSI-LD. Quando si parla di "contestualizzazione", si fa riferimento alla capacità di comprendere e interpretare il mondo che ci circonda attraverso una vasta gamma di dati e informazioni. Orion-LD è progettato per fare proprio questo, fornendo un'interfaccia standard e unificata per accedere e manipolare il contesto in modi significativi e utili. Utilizzando un modello dati basato su Linked Data, Orion-LD collega e arricchisce i dati provenienti da diverse fonti, consentendo di scoprire e sfruttare nuove relazioni e connessioni tra i dati. Una delle caratteristiche più potenti di Orion-LD è la sua capacità di notificare le applicazioni quando cambia lo stato di un determinato oggetto o entità nel contesto. Questo meccanismo di notifiche e sottoscrizioni consente alle applicazioni di rimanere sincronizzate con il mondo reale e di reagire istantaneamente ai cambiamenti, consentendo una vasta gamma di scenari di utilizzo, come il monitoraggio ambientale, la gestione del traffico, la sicurezza pubblica e molto altro. A livello architetturale, Orion-LD si appoggia a MongoDB, come database NoSQL, per memorizzare le informazioni relative allo stato più recente del contesto.

Modellazione dati per il contesto di Orion-LD

In questa sezione viene presentato il data modeling pensato per rappresentare il contesto che Orion-LD deve avere. Per partire con questa modellazione, di seguito, in Figura 3.15 viene mostrato il modello ER (Entity-Relationship)

realizzato per trattare il dominio del problema in cui rientra il progetto REMOTE.

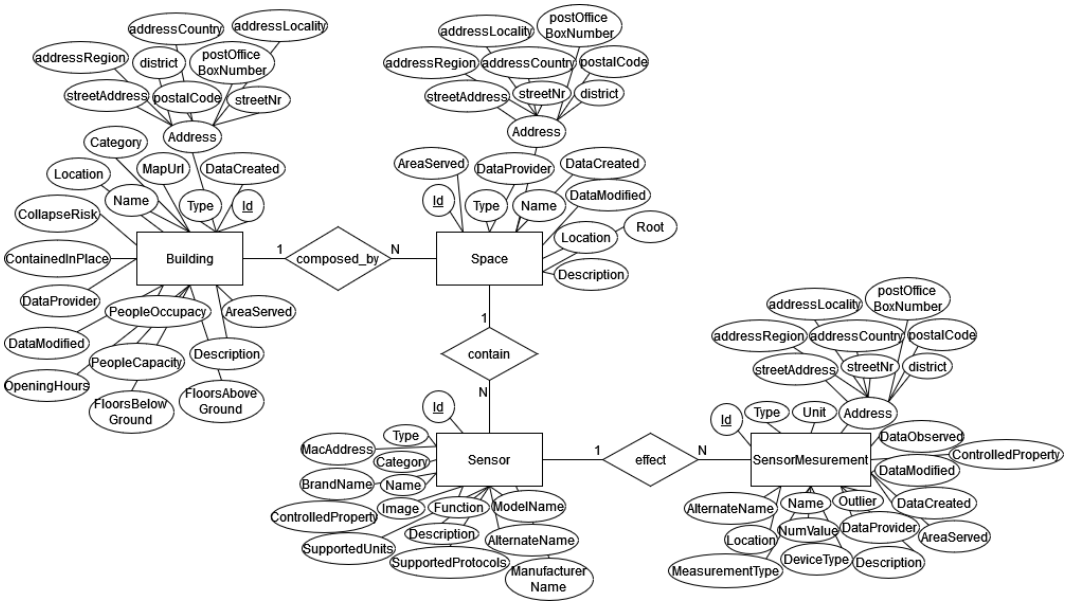

Figure 3.15: Modello ER

Successivamente, una volta definito il modello ER, serve scegliere quali data model utilizzare per rappresentare le entità da creare ed utilizzare all'interno di Orion-LD. Per questo progetto, è stato deciso di fare riferimento agli *Smart Data Models* [3] che vengono ampiamente utilizzati nell'ecosistema FIWARE. Nello specifico, si è deciso di utilizzare lo smart data model *BuldingType* [4] per far riferimento alle entità Building e Space del modello ER dato che dà la possibilità di definire una gerarchia di entità per rappresentare nel miglior modo la gerarchie tra i vari spazi che compongono una struttura (per spazi si possono intendere anche altri componenti che non facciano strettamente riferimento ad una stanza o altro) in maniera elastica. Infine, per le entità Sensor e SensorMeasurement si è deciso di adottare gli smart data models *DeviceModel* e *DeviceMeasurement* [5]. Il primo, permette di definire in maniera generica un dispositivo di sensing, utile per lo scopo del progetto dato che viene simulato il comportamento di un sensore, raggruppando le sue caratteristiche principali, senza entrare nello specifico (ad es. dal punto di vista dell'hardware, firmware e software che lo compongono), cosa che fa lo smart data model *Device*. Il secondo modello invece, permette di definire le caratteristiche riguardanti una misurazione effettuata da un sensore, che è perfettamente in linea con quello proposto sopra nel modello ER.

3.10.6 QuantumLeap

QuantumLeap è un servizio REST per memorizzare, interrogare e recuperare dati spazio-temporali NGSI-v2 e NGSI-LD (attualmente ancora con supporto sperimentale). QuantumLeap converte i dati semi-strutturati NGSI in formato

tabellare e li memorizza in un database timeseries, associando ciascun record del database con un indice temporale e, se presente nei dati NGSI, una posizione sulla Terra o in coordinate relative. I client REST possono quindi recuperare le entità NGSI filtrando gli insiemi di entità attraverso intervalli temporali e operatori spaziali. Si noti che, dal punto di vista del client, queste query sono definite sulle entità NGSI anziché sulle tabelle del database. Tuttavia, la funzionalità di query disponibile tramite l'interfaccia REST è piuttosto basilare e la maggior parte delle query complesse richiede tipicamente ai client di utilizzare direttamente il database. La specifica dell'API REST, chiamata **NGSI-TSDB**, che QuantumLeap implementa, è stata definita con l'obiettivo di fornire un'interfaccia REST agnostica rispetto al database per la memorizzazione, l'interrogazione e il recupero di serie temporali di entità NGSI che potrebbero essere il più possibile vicine alla specifica NGSI stessa. Pertanto, NGSI-TSDB fornisce un meccanismo uniforme e familiare per accedere ai dati delle serie temporali che consente di implementare servizi come QuantumLeap per supportare in modo trasparente più backend di database. Infatti, attualmente QuantumLeap supporta sia **CrateDB** che **Timescale** come database di backend. Dal punto di vista architetturale, tipicamente QuantumLeap acquisisce dati IoT sotto forma di entità NGSI, da un livello di IoT Agent, indirettamente (**in questo progetto non si ha il livello IoT Agent dato che si sta supponendo che il sensore simulato sia FIWARE-Ready**), tramite notifiche NGSI impostate in anticipo con il context broker, Orion-LD. Come già accennato, le entità NGSI in ingresso vengono convertite in record di database e memorizzate in uno dei backend di database di serie temporali configurati. Spesso vengono deployati anche strumenti di visualizzazione come Grafana al fine di visualizzare i dati di serie temporali memorizzati nel database, cosa che viene fatta nel progetto REMOTE. La Figura 3.16 illustra le relazioni e le interazioni tra questi componenti in uno scenario tipico di deployment di QuantumLeap.

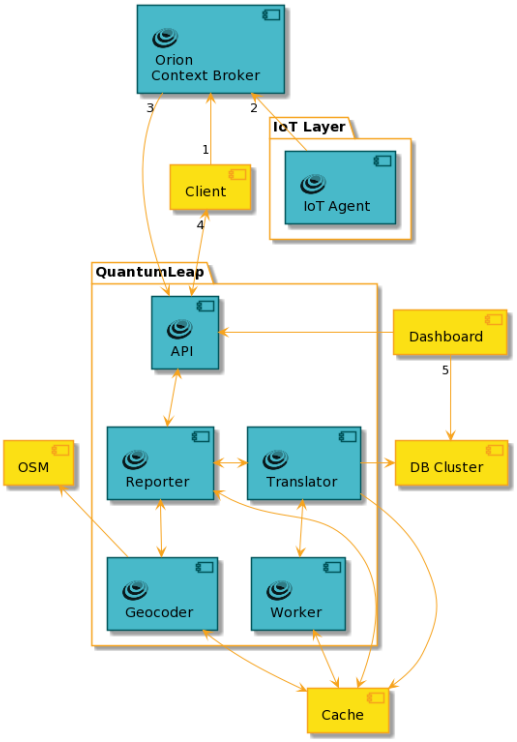

Figure 3.16: Architettura di Quantumleap [6]

CrateDB è il database di backend predefinito. È facile da scalare grazie alla sua architettura shared-nothing che si presta bene alla containerizzazione. Inoltre, CrateDB utilizza SQL per interrogare i dati, con estensioni integrate per le query temporali e geografiche. CrateDB offre anche un'API Postgres, semplificando la sua integrazione. Ad esempio, è possibile sfruttare il plugin PostgreSQL di Grafana per visualizzare le serie temporali memorizzate in CrateDB. QuantumLeap memorizza le entità NGSI in CrateDB utilizzando l'endpoint notify. Per questi motivi si è deciso di adottare CrateDB come database di backend, affiancato da Grafana per la visualizzazione dei dati immagazzinati, come già accennato in precedenza.

3.10.7 Perseo

Perseo è un software di Complex Event Processing (*CEP*) basato su *Esper*, progettato per essere pienamente conforme a NGSI-v2. Utilizza NGSI-v2 come protocollo di comunicazione per gli eventi e, quindi, Perseo è in grado di lavorare in modo trasparente e integrato con i context broker. Il context broker testato con Perseo e ufficialmente supportato è Orion e Orion-LD di conseguenza. Il suo funzionamento, Figura 3.17, si basa sull'ascoltare gli eventi provenienti dalle informazioni di contesto per identificare schemi descritti da regole, al fine di reagire immediatamente ad essi attivando azioni. Sfruttando il meccanismo di notifiche, i client istruiscono Orion-LD a notificare a Perseo i cambiamenti nelle entità di loro interesse (Event API). Successivamente, le regole per il motore CORE, possono essere facilmente gestite utilizzando qual-

siasi dei client REST (Postman, curl, ecc.) in grado di utilizzare in modo programmatico l'API delle Regole di Perseo. Queste regole identificheranno schemi che attiveranno azioni con Orion-LD per creare o aggiornare entità, o con altri componenti o sistemi esterni diversi, come Web-server (HTTP), Email (SMTP), SMS (SMPP) o eventualmente anche *X* (Ex *Twitter*). Pertanto, Perseo si presta perfettamente alla necessità che si ha nel progetto REMOTE, di analizzare i cambiamenti che si verificano su determinate entità, come i valori misurati dai sensori, e di applicare le regole preimpostate, se soddisfatte per avvisare l'utente in tempo reale.

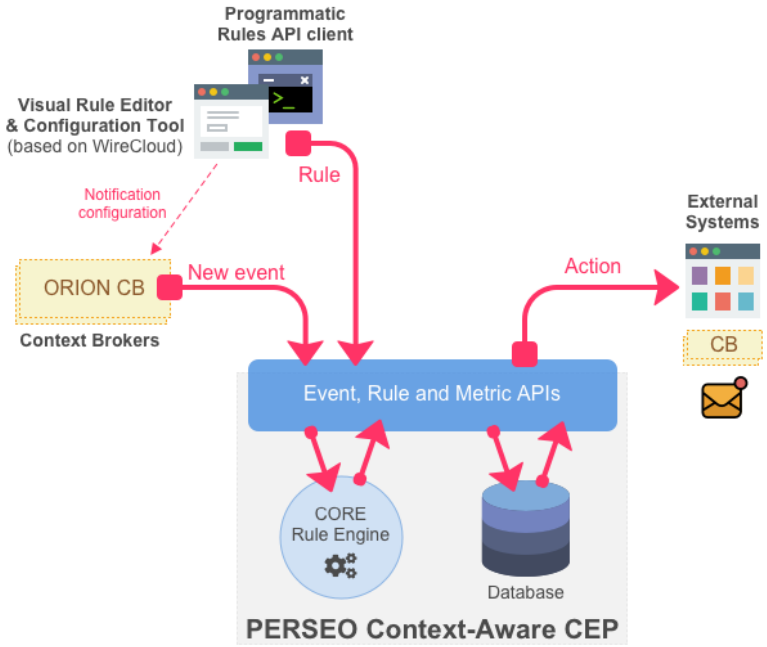

Figure 3.17: Architettura di PERSEO [6]

A livello architetturale è formato dai seguenti componenti, Figura 3.18:

- **Perseo-fe**: Il "front-end" del CEP è responsabile del processing degli eventi in arrivo e delle regole, memorizzando le regole ed eseguendo le azioni. L'esecuzione delle azioni viene registrata per controllare la frequenza delle azioni eseguite. Inoltre, controlla le regole di non aggiornamento ("no-signal") che impostano un intervallo di tempo massimo tra gli eventi provenienti da un'entità del broker di contesto;
- **Core (Perseo-Core)**: Il "back-end" del CEP, il "rule engine". Esamina gli eventi in arrivo rispetto alle regole in *EPL* (Event Processing Language) e invoca Perseo se deve essere eseguita un'azione. Non dispone di archiviazione persistente. Le regole sono mantenute in memoria. L'intero set di regole viene periodicamente aggiornato dal "front-end" di Perseo;
- **MongoDB**: Il database utilizzato da Perseo per memorizzare regole ed esecuzioni di azioni.

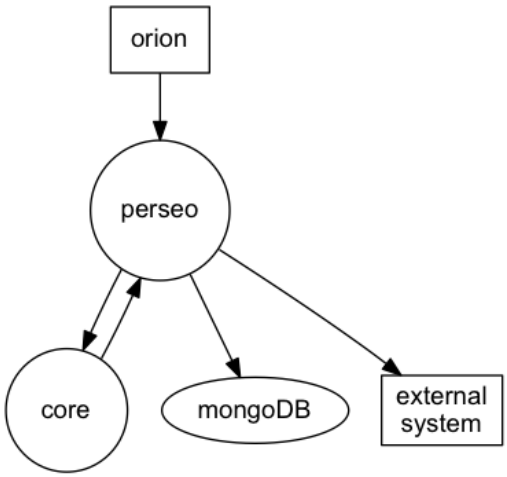

Figure 3.18: Architettura di PERSEO (2) [6]

3.10.8 Component diagram

I diagrammi dei componenti contengono, come dice il nome stesso, dei componenti che rappresentano pezzi di software che possono essere acquisiti e aggiornati in modo indipendente. Essi hanno a che fare con la modalità con cui gli utenti interagiscono con il software. I collegamenti tra ogni singolo componente sono indicati dalle interfacce che possono essere implementate o richieste [1].
In Figura 3.19 viene mostrato il component diagram di questo prototipo di applicazione.

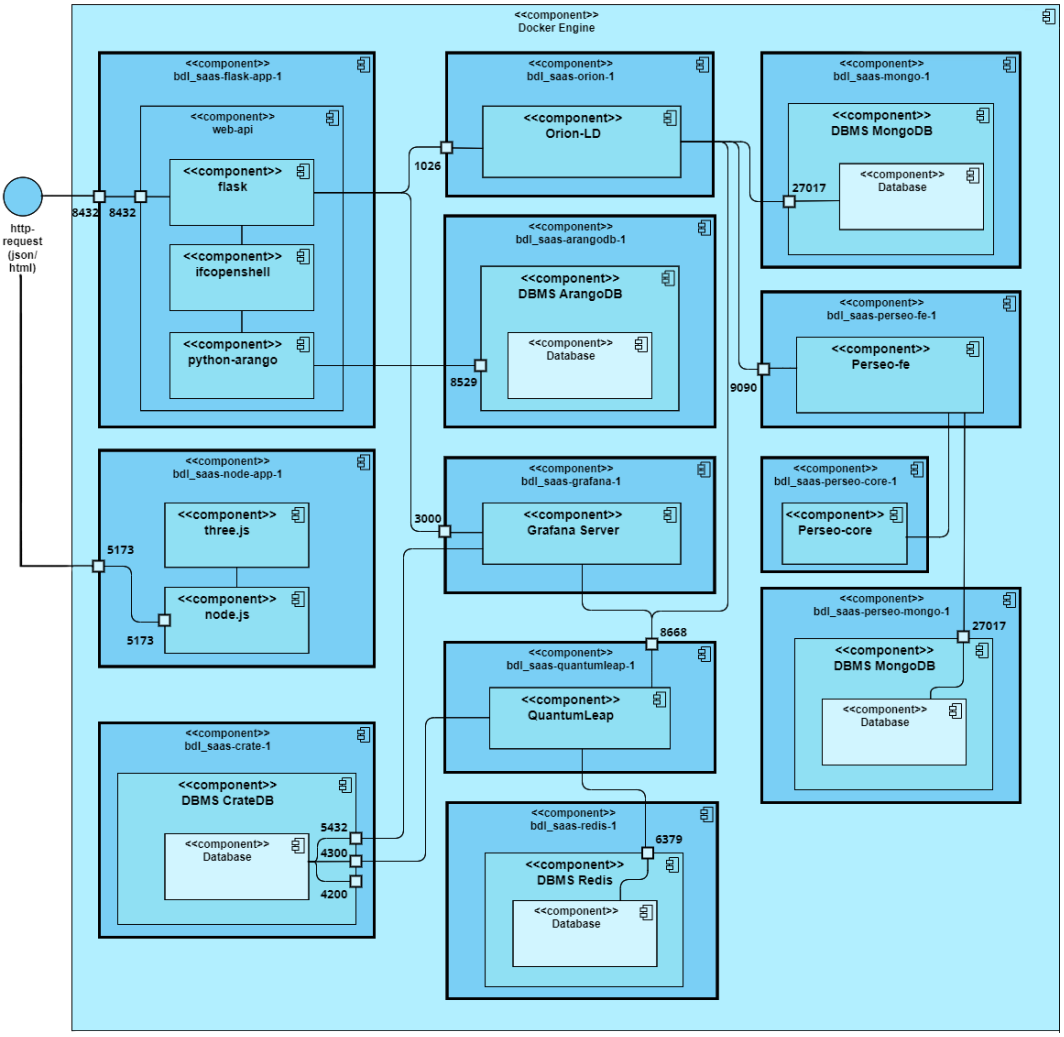

Figure 3.19: Component diagram

Implementazione

4.1 Introduzione

In questo capitolo, verrà mostrata l'implementazione concreta di un caso d'uso tra quelli presentati nella sezione 3.1.

4.2 Implementazione dell'algoritmo di mapping

L'algoritmo che verrà mostrato in questa sezione si basa sulla falsa riga da quello proposto in "*IFC-graph for facilitating building information access and query*" [2] con adattamenti ed alcune modifiche per l'utilizzo su ArangoDB. Per l'implementazione è stato creato uno script Python che utilizza le seguenti librerie:

- **python-arango**: libreria che fornisce un'interfaccia Python per interagire con il database ArangoDB. È utilizzata per stabilire la connessione al database, gestire le operazioni CRUD e eseguire query AQL. (versione utilizzata: 7.9.1);
- **ifcopenshell**: utilizzata per il parsing e l'estrazione di informazioni dettagliate dai file IFC. (versione utilizzata: 0.7.0.231218);
- **uuid**: libreria per la creazione di un identificatore casuale di testo. (versione utilizzata: 1.26.4).
- **lark-parser**: libreria Python utilizzata da ifcopenshell per il parsing dei dati nel documento IFC.(versione utilizzata: 0.12.0).

Si faccia riferimento dalla sezione 3.5 in poi, per i dettagli e le ragioni implementative legate ai vari metodi. Di seguito il listato del codice [7]:

```
1  def create_pure_node_from_ifc_entity(ifc_entity):
2      key = f"{ifc_entity.is_a()}-{str(ifc_entity.id() if ifc_entity.id() != 0 else uuid4())}"
3      node = {'_key': key, 'name': ifc_entity.is_a()}
4
5      attributes_type = ['ENTITY INSTANCE','AGGREGATE OF ENTITY INSTANCE', 'DERIVED']
6
7      for i in range(ifc_entity.__len__()):
8          if not ifc_entity.wrapped_data.get_argument_type(i)
9              in attributes_type:
10
11             name = ifc_entity.wrapped_data.get_argument_name(i)
12             name_value = ifc_entity.wrapped_data.get_argument(i)
13             node[name] = name_value
```

```
    return node

def create_graph_from_ifc_entity_all(db, ifc_entity, ifc_file, nodes_name, edges_name):
    node = create_pure_node_from_ifc_entity(ifc_entity)
    node_key = node["_key"]
    if node_key not in existing_nodes:
        collection = db[nodes_name]
        collection.insert(node)
        existing_nodes.add(node_key)

    for i in range(ifc_entity.__len__()):
        if ifc_entity[i]:
            if ifc_entity.wrapped_data.get_argument_type(i) == 'ENTITY INSTANCE':
                if ifc_entity.is_a() in ['IfcOwnerHistory'] and ifc_entity.is_a() != 'IfcProject':
                    continue
                else:
                    sub_node = create_pure_node_from_ifc_entity(ifc_entity[i])
                    sub_node_key = sub_node["_key"]
                    if sub_node_key not in existing_nodes:
                        sub_collection = db[nodes_name]
                        sub_collection.insert(sub_node)
                        existing_nodes.add(sub_node_key)

                    edge_key = (node["_key"], sub_node["_key"],
                        ifc_entity.wrapped_data.get_argument_name(i))

                    if edge_key not in existing_edges:
                        rel_collection = db[edges_name]
                        rel_collection.insert({
                            "_from": f"{nodes_name}/{node['_key']}",
                            "_to": f"{nodes_name}/{sub_node['_key']}",
                            "rel_name": ifc_entity.wrapped_data.get_argument_name(i)
                        })
                        existing_edges.add(edge_key)

            elif ifc_entity.wrapped_data.get_argument_type(i) == 'AGGREGATE OF ENTITY INSTANCE':
                for sub_entity in ifc_entity[i]:
                    sub_node = create_pure_node_from_ifc_entity(sub_entity)
                    sub_node_key = sub_node["_key"]
                    if sub_node_key not in existing_nodes:
                        sub_collection = db[nodes_name]
                        sub_collection.insert(sub_node)
                        existing_nodes.add(sub_node_key)

                    edge_key = (node["_key"], sub_node["_key"],
                        ifc_entity.wrapped_data.get_argument_name(i))

                    if edge_key not in existing_edges:
                        rel_collection = db[edges_name]
                        rel_collection.insert({
                            "_from": f"{nodes_name}/{node['_key']}",
                            "_to": f"{nodes_name}/{sub_node['_key']}",
                            "rel_name": ifc_entity.wrapped_data.get_argument_name(i)
                        })
                        existing_edges.add(edge_key)

    for rel_name in ifc_entity.wrapped_data.get_inverse_attribute_names():
        if ifc_entity.wrapped_data.get_inverse(rel_name):
            inverse_relations = ifc_entity.wrapped_data.get_inverse(rel_name)
            for wrapped_rel in inverse_relations:
                rel_entity = ifc_file.by_id(wrapped_rel.id())
                sub_node = create_pure_node_from_ifc_entity(rel_entity)
                sub_node_key = sub_node["_key"]
                if sub_node_key not in existing_nodes:
                    sub_collection = db[nodes_name]
                    sub_collection.insert(sub_node)
                    existing_nodes.add(sub_node_key)
                edge_key = (sub_node["_key"], sub_node["_key"], rel_name)
                if edge_key not in existing_edges:
                    rel_collection = db[edges_name]
                    rel_collection.insert({
                        "_from": f"{nodes_name}/{sub_node['_key']}",
                        "_to": f"{nodes_name}/{sub_node['_key']}",
                        "rel_name": rel_name
                    })
                    existing_edges.add(edge_key)

def create_full_graph(db, ifc_file, nodes_name, edges_name):
    idx = 1
    length = len(ifc_file.wrapped_data.entity_names())
    for entity_id in ifc_file.wrapped_data.entity_names():
        entity = ifc_file.by_id(entity_id)
        print(idx, '/', length, entity)
        create_graph_from_ifc_entity_all(db, entity, ifc_file, nodes_name, edges_name)
        idx += 1
    return
```

4.3 Implementazione dell'API per la comunicazione con i componenti Fiware

Nonostante l'esistenza delle API fornite da FIWARE [6] per la comunicazione con il Context Broker, il CEP e QuantumLeap, si è optato per lo sviluppo di un'API Python aggiuntiva per avere dei metodi che permettessero di formattare sia l'header che il payload da mandare agli endpoint dei componenti di FIWARE. Questa API implementa metodi distinti per operazioni come le sottoscrizioni, inserimento di regole e per il ciclo CRUD delle entità all'interno dei componenti.
Di seguito, alcuni esempi di codice [7] per la creazione di una sottoscrizione ad Orion-LD, di una regola per Perseo-fe e l'inserimento di un'entità tramite l'API FIWARE attraverso una POST:

```python
def init_quantumleap_subscription(self, description, type, format, uri):
    json = self.get_subscriptions()

    for sub in json:
        if sub["notification"]["endpoint"]["uri"] == uri and sub["entities"] == type:
            self.delete_subscription(sub["id"])

    payload = {...}
    res = self.subscribe(payload)
    return res

def init_perseo_subscription(self, description, type, format, uri):
    json = self.get_subscriptions()

    for sub in json:
        if sub["notification"]["endpoint"]["uri"] == uri and sub["entities"] == type:
            self.delete_subscription(sub["id"])

    payload = {...}
    res = self.subscribe(payload)
    return res
```

```python
def init_rules_mail(self, rule_name, text, template, to, subject):
    status = self.get_rule_by_name(rule_name)

    if status == 200:
        self.delete_rule(rule_name)

    payload = {
        "name": rule_name,
        "text": text,
        "action": {
            "type": "email",
            "template": template,
            "parameters": {
                "to": to,
                "from": "perseobdl@gmail.com",
                "subject": subject
            }
        }
    }

    res = self.insert_rule(payload)
    return res
```

```python
def insert_entity(self, data):
    url = f'http://{self.orionIP}/ngsi-ld/v1/entities'
    response = requests.post(url, headers=self.header, json=data)
    return response.status_code
```

Per quanto riguarda il contesto da passare, si è deciso di basarsi su quelli associati ai vari Smart Data Models [3] e di inserirlo nel payload delle varie richieste effettuate, da come si può vedere nel codice sotto. Da precisare inoltre, come già detto nella parte di Progettazione, che **i dati che vengono mandati a QuantumLeap sono di tipo FIWARE-Ready pertanto si è deciso di non utilizzare un IoT Agent, dato che si tratta di dati simulati**.

```
self.header = {
    'Content-Type': 'application/ld+json'
}
self.header_subscription = {
    'Content-Type': 'application/ld+json',
    'Accept': 'application/ld+json'
}
self.header_perseo = {
    'Content-Type': 'application/json'
}
self.header_perseo_subscription = {
    'Content-Type': 'application/json',
    'Accept': 'application/json'
}
```

4.4 Implementazione visualizzatore 3D

Per quanto riguarda la visualizzazione del modello 3D, il frame HTML utilizzato per contenere il visualizzatore, fa uso di uno script TypeScript (con libreria Three.js) per la renderizzazione della scena e dei modelli 3D estratti dai file IFC. Successivamente viene mostrato il listato contenente una parte del codice implementato [7] per la realizzazione del visualizzatore. Come si può vedere dagli import, la libreria **openbim-components** [8] è stata principalmente utile per effettuare il caricamento di un file IFC, estrarre il relativo modello .glb, effettuarne il rendering, ottimizzarlo e creare la scena da visualizzare. Oltre a queste operazioni, la libreria permette di utilizzare strumenti come il raycaster e l'highlighter per evidenziare i componenti del modello selezionati con il mouse e strumenti per realizzare toolbar, eliminare i modelli caricati e estrarre le proprietà "hard-coded" presenti all'interno del file IFC, come i vari property set. Successivamente la libreria **jsoneditor** [9] ha permesso di realizzare l'interfaccia per la visualizzazione dei risultati delle varie query AQL scelte dall'utente, sotto forma di tree. Oltre a queste cose, nel listato sono stati implementati i vari metodi che permettono di ottenere i risultati delle query selezionate invocando rispettivamente i vari endpoint messi a disposizione da Flask, grazie anche a **flask-cors**, con la conseguente formattazione dei dati ottenuti in formato JSON.

```
import * as OBC from "openbim-components";
import * as THREE from "three";
import * as JSONEditor from 'jsoneditor';

const hostIPAddress = window.location.hostname;
const viewer = new OBC.Components();
viewer.onInitialized.add(() => {
});

const sceneComponent = new OBC.SimpleScene(viewer);
sceneComponent.setup();
viewer.scene = sceneComponent;

const viewerContainer = document.getElementById(
    "webinar-sharepoint-viewer"
) as HTMLDivElement;
const rendererComponent = new OBC.PostproductionRenderer(
    viewer,
    viewerContainer
);
viewer.renderer = rendererComponent;
const postproduction = rendererComponent.postproduction;

const cameraComponent = new OBC.OrthoPerspectiveCamera(viewer);
viewer.camera = cameraComponent;

const raycasterComponent = new OBC.SimpleRaycaster(viewer);
viewer.raycaster = raycasterComponent;

viewer.init();
postproduction.enabled = true;
```

```
32
33 const grid = new OBC.SimpleGrid(viewer, new THREE.Color(0x666666));
34 postproduction.customEffects.excludedMeshes.push(grid.get());
35
36 const ifcLoader = new OBC.FragmentIfcLoader(viewer);
37
38 ifcLoader.settings.wasm = {
39     absolute: true,
40     path: "https://unpkg.com/web-ifc@0.0.44/",
41 };
42
43 const ifcManager = new OBC.FragmentManager(viewer);
44 ifcManager.uiElement.get("main").materialIcon = "delete";
45 ifcManager.uiElement.get("main").tooltip = "Remove models";
46
47
48 const highlighter = new OBC.FragmentHighlighter(viewer);
49 highlighter.setup();
50
51 const propertiesProcessor = new OBC.IfcPropertiesProcessor(viewer);
52 highlighter.events.select.onClear.add(() => {
53     propertiesProcessor.cleanPropertiesList();
54 });
55
56
57 ifcLoader.onIfcLoaded.add(async (model) => {
58     propertiesProcessor.process(model);
59     highlighter.clear();
60     highlighter.events.select.onHighlight.add(async (selection) => {
61         const fragmentID = Object.keys(selection)[0];
62         const expressID = Number([...selection[fragmentID]][0]);
63         propertiesProcessor.renderProperties(model, expressID);
64
65         //Function for query information about selected component
66         try {
67             const modelName = ifcManager.groups[0].name;
68             if (modelName) {
69                 fetch(`http://${hostIPAddress}:8432/get_node_by_id/` + modelName.split(".")[0] + '_nodes/' + expressID)
70                     .then(response => {
71                         if (!response.ok) {
72                             throw new Error('Network response was not ok');
73                         }
74                         return response.json();
75                     })
76                     .then(data => {
77                         const jsonContainer = document.getElementById("jsoneditor");
78                         if (jsonContainer) {
79                             while (jsonContainer.firstChild) {
80                                 jsonContainer.removeChild(jsonContainer.firstChild);
81                             }
82                             const options = {};
83                             const editor = new JSONEditor(jsonContainer, options);
84                             editor.set(data);
85                         }
86                     })
87                     .catch(error => {
88                         const jsonContainer = document.getElementById("jsoneditor");
89                         if (jsonContainer) {
90                             while (jsonContainer.firstChild) {
91                                 jsonContainer.removeChild(jsonContainer.firstChild);
92                             }
93                             const options = {};
94                             const editor = new JSONEditor(jsonContainer, options);
95                             editor.set({"error": "Please go back ad start mapping procedure with file insert here"});
96                         }
97                     });
98             } else {
99                 const jsonContainer = document.getElementById("jsoneditor");
100                 if (jsonContainer) {
101                     while (jsonContainer.firstChild) {
102                         jsonContainer.removeChild(jsonContainer.firstChild);
103                     }
104                     const options = {};
105                     const editor = new JSONEditor(jsonContainer, options);
106                     editor.set({"Error": "No model found in scene"});
107                 }
108             }
109         } catch (error) {
110             const jsonContainer = document.getElementById("jsoneditor");
111             if (jsonContainer) {
112                 while (jsonContainer.firstChild) {
113                     jsonContainer.removeChild(jsonContainer.firstChild);
114                 }
115                 const options = {};
116                 const editor = new JSONEditor(jsonContainer, options);
117                 editor.set({"error": error});
118             }
119         }
120     });
121     highlighter.update();
122
123 });
124
125
126 const length = new OBC.LengthMeasurement(viewer);
```

```
length.enabled = true;
length.snapDistance = 1;


const queryTool = new OBC.Button(viewer);
queryTool.materialIcon = "search";
queryTool.tooltip = "Query Tool";

const allNodesButton = new OBC.Button(viewer);
const allEdgesButton = new OBC.Button(viewer);
const traversalNodeButton = new OBC.Button(viewer);
const nodesByTypeButton = new OBC.Button(viewer);

allNodesButton.onClick.add(() => fetchAllNodes());
allEdgesButton.onClick.add(() => fetchAllEdges());

allNodesButton.label = "Show all nodes";
allEdgesButton.label = "Show all edges";
traversalNodeButton.label = "Show node details";
nodesByTypeButton.label = "Show nodes by type";

const addSensorButton = new OBC.Button(viewer);
addSensorButton.materialIcon = "sensors";
addSensorButton.tooltip = "Add sensor";
addSensorButton.onClick.add(() => showSensorForm());

const goBackButton = new OBC.Button(viewer);
goBackButton.materialIcon = "exit_to_app";
goBackButton.tooltip = "Go back";

goBackButton.onClick.add(() => redirectTo(`http://${hostIPAddress}:8432/menu`));

queryTool.addChild(allNodesButton);
queryTool.addChild(allEdgesButton);
queryTool.addChild(traversalNodeButton);
queryTool.addChild(nodesByTypeButton);
traversalNodeButton.onClick.add(() => showTraversalFields());
nodesByTypeButton.onClick.add(() => showTraversalByTypeFields());

const refreshButton = new OBC.Button(viewer);
refreshButton.materialIcon = "refresh";
refreshButton.tooltip = "Reset model";
refreshButton.onClick.add(() => {
    window.location.reload();
});

const mainToolbar = new OBC.Toolbar(viewer);
mainToolbar.addChild(
    ifcLoader.uiElement.get("main"),
    refreshButton,
    propertiesProcessor.uiElement.get("main"),
    queryTool,
    addSensorButton,
    goBackButton
);
viewer.ui.addToolbar(mainToolbar);

window.addEventListener("thatOpen", async (event: any) => {
    const {name, payload} = event.detail;
    if (name === "openModel") {
        const {name, buffer} = payload;
        const model = await ifcLoader.load(buffer, name);
        const scene = viewer.scene.get();
        scene.add(model);
    }
});

window.onkeydown = (event) => {
    if (event.code === "Delete" || event.code === "Backspace") {
        length.delete();
    }
};

window.addEventListener("mousedown", function (event) {
    if (event.button === 1) { // Il pulsante della rotellina del mouse ha il codice 1
        // Attiva la misurazione
        length.create();
        // generateTable(jsonData);
    }
});

function redirectTo(url: string): void {...}
function showTraversalFields() {...}
function showTraversalByTypeFields() {...}
function fetchAllEdges() {...}
function fetchTraversal() {...}
function fetchTraversalByName() {...}
function showSensorForm() {...}
function createSensor() {...}
```

Questa implementazione permette di integrare in modo interattivo i modelli 3D all'interno dell'interfaccia utente, come si vede in Figura 4.1 e 4.2:

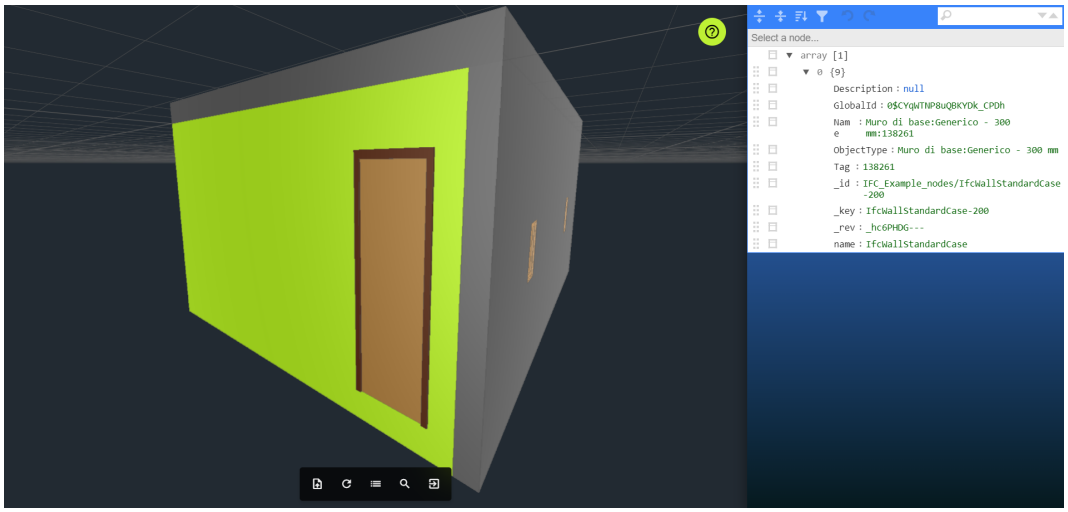

Figure 4.1: Visualizzatore del modello 3D

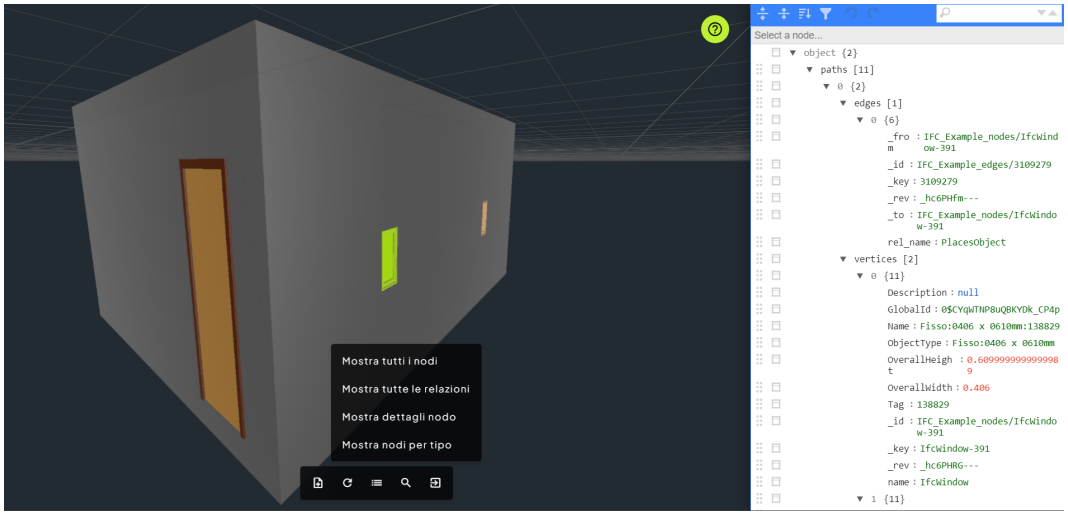

Figure 4.2: Esempio di query sul modello

4.5 Implementazione dell'interfaccia utente

Per l'interfaccia utente sono stati utilizzati HTML e CSS per la creazione dei template renderizzati dalle route di Flask. Nelle Figure 4.3 e 4.4 alcuni esempi:

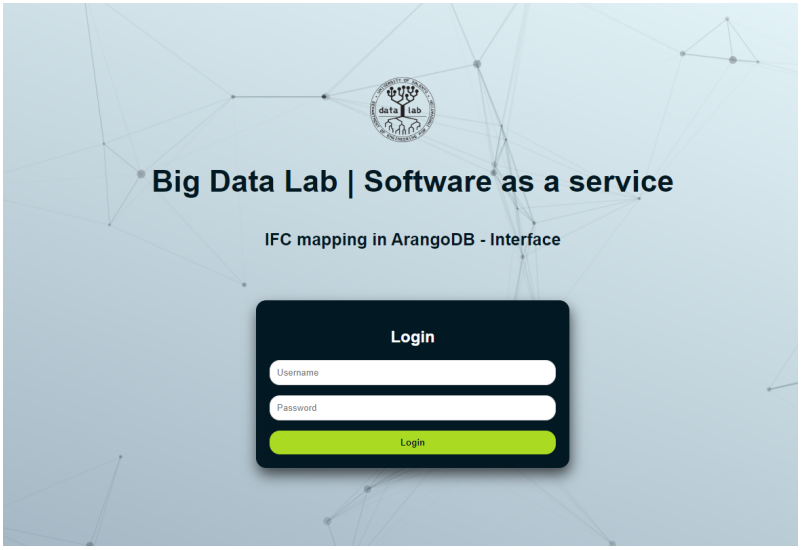

Figure 4.3: Interfaccia di login

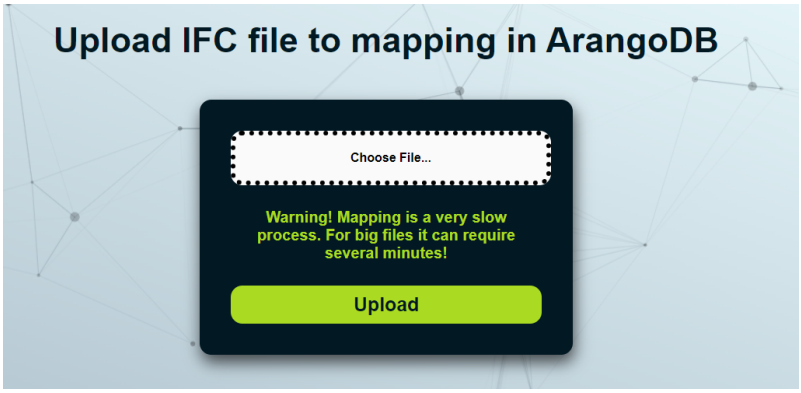

Figure 4.4: Interfaccia per il caricamento del file IFC

4.6 Implementazione dei file Docker

Per la dockerizzazione del progetto bisogna creare il file docker-compose per la gestione dei container Docker necessari per l'applicazione, oltre ai relativi Dockerfile per la parte di progetto realizzata con Flask e quella realizzata con Node.js.
Nel codice visibile sotto [7] vengono definiti i servizi, le reti e i volumi necessari per eseguire l'applicazione, insieme alle porte esposte da ciascun container per la comunicazione sia interna che esterna ed ovviamente le varie versioni adottate per ogni servizio, da precisare che i campi tra parentesi angolari <> sono

modificabili a piacere a seconda delle esigenze e che per l'app flask è stato fatto un binding diretto sulla cartella del progetto Python pertanto è eventualmente da adattare il percorso a seconda di dove viene inserita la cartella relativa:

```
---
#Dockerfile Flask App
FROM python:3.11

COPY requirements.txt .

RUN \
    python3 -m pip install -r requirements.txt --no-cache-dir

COPY . ./app

EXPOSE 8432

ENV FLASK_APP=/app/src/App.py
ENV FLASK_ENV=development

CMD ["flask", "run", "--host=0.0.0.0", "--port=80", "--debug"]
---
```

```
---
#Dockerfile Node App
FROM node:21.6.1

WORKDIR /app

COPY package*.json .
COPY . .

RUN npm install

EXPOSE 5173

CMD ["npm", "run", "dev"]
---
```

```
---
#docker-compose
version: '3.9'

services:
  flask_app:
    build:
      context: ./flask_root
      dockerfile: Dockerfile
    ports:
      - "8432:80"
    environment:
      - DEBUG_USR=admin
      - DEBUG_PWD=restapi
    volumes:
```

```
16        - ./flask_root:/app
17      networks:
18        - general_network
19
20    node_app:
21      build:
22        context: ./node_root
23        dockerfile: Dockerfile
24      ports:
25        - "5173:5173"
26      volumes:
27        - node_app:/app
28      networks:
29        - general_network
30
31    arangodb:
32      image: arangodb:3.11.7
33      ports:
34        - "8529:8529"
35      environment:
36        ARANGO_ROOT_PASSWORD: "BDLaaS"
37      volumes:
38        - arangodb_data:/var/lib/arangodb3
39        - arangodb_apps:/var/lib/arangodb3-apps
40        - arangodb_backup:/var/lib/arangodb3-backup
41      networks:
42        - general_network
43
44    orion:
45      image: fiware/orion-ld:1.6.0-PRE-1578
46      ports:
47        - "1026:1026"
48      command: -dbhost mongo -logLevel DEBUG
49      depends_on:
50        - mongo
51      healthcheck:
52        test: curl --fail -s http://orion:1026/version || exit 1
53      networks:
54        - general_network
55
56    quantumleap:
57      image: orchestracities/quantumleap:1.0.0
58      ports:
59        - "8668:8668"
60      depends_on:
61        - mongo
62        - orion
63        - crate
64      environment:
65        - CRATE_HOST=${CRATE_HOST:-crate}
66        - USE_GEOCODING=True
67        - REDIS_HOST=redis
68        - REDIS_PORT=6379
69        - LOGLEVEL=DEBUG
70      networks:
71        - general_network
```

```
72
73    mongo:
74      image: mongo:${MONGO_VERSION:-4.4}
75      ports:
76        - "27017:27017"
77      volumes:
78        - mongodata:/data/db
79        - mongoconfig:/data/configdb
80        - mongolog:/var/log/mongodb
81      networks:
82        - general_network
83
84    crate:
85      image: crate:${CRATE_VERSION:-5.6.2}
86      command: crate -Cauth.host_based.enabled=false
87        -Ccluster.name=democluster -Chttp.cors.enabled=true -Chttp.cors.allow-origi
88      environment:
89        - LOG4J_FORMAT_MSG_NO_LOOKUPS=true
90      ports:
91        # Admin UI
92        - "4200:4200"
93        # Transport protocol
94        - "4300:4300"
95        - "5432:5432"
96      volumes:
97        - cratedata:/data
98      networks:
99        - general_network
100
101   grafana:
102     image: grafana/grafana
103     ports:
104       - "3000:3000"
105     depends_on:
106       - crate
107     environment:
108       - GF_SECURITY_ALLOW_EMBEDDING=true
109       - GF_AUTH_ANONYMOUS_ENABLED=true
110       - GF_AUTH_ORG_ROLE=viewer
111     volumes:
112       - grafana_data:/var/lib/grafana
113       - grafana_logs:/var/log/grafana
114     networks:
115       - general_network
116
117   redis:
118     image: redis:${REDIS_VERSION:-7.2.4}
119     ports:
120       - "6379:6379"
121     volumes:
122       - redisdata:/data
123     networks:
124       - general_network
125
126   perseo-mongo:
127     image: mongo:4.4
```

```
    volumes:
      - perseomongodata:/data/db
      - perseomongoconfig:/data/configdb
      - perseomongolog:/var/log/mongodb
    networks:
      - general_network
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure

  perseo-core:
    image: fiware/perseo-core:1.13.0
    environment:
      - PERSEO_FE_URL=http://perseo-fe:9090
      - MAX_AGE=6000
    networks:
      - general_network
    depends_on:
      - perseo-mongo
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure

  perseo-fe:
    image: fiware/perseo:1.26.0
    networks:
      - general_network
    ports:
      - "9090:9090"
    depends_on:
      - perseo-core
    environment:
      - PERSEO_MONGO_ENDPOINT=perseo-mongo
      - PERSEO_CORE_URL=http://perseo-core:8080
      - PERSEO_LOG_LEVEL=debug
      - PERSEO_ORION_URL=http://orion:1026/
      - PERSEO_SMTP_HOST=smtp.gmail.com
      - PERSEO_SMTP_PORT=465
      - PERSEO_SMTP_SECURE=true
      - PERSEO_SMTP_AUTH_USER=perseobdl@gmail.com
      - PERSEO_SMTP_AUTH_PASS=ybxdpminubbnlcka
    deploy:
      replicas: 1
      restart_policy:
        condition: on-failure

volumes:
  node_app:
  arangodb_data:
  arangodb_apps:
  arangodb_backup:
  mongodata:
  mongoconfig:
  mongolog:
```

```
184     perseomongodata:
185     perseomongoconfig:
186     perseomongolog:
187     cratedata:
188     redisdata:
189     grafana_data:
190     grafana_logs:
191
192 networks:
193     general_network:
194 ---
```

Bibliografia

[1] Martin Fowler. *UML Distilled - Guida rapida al linguaggio di modellazione standard*. Pearson, 2018.

[2] Junxiang Zhu, Peng Wu, and Xiang Lei. Ifc-graph for facilitating building information access and query. *Automation in Construction*, 148, 2023.

[3] Smart data models. `https://www.fiware.org/smart-data-models/`.

[4] Smart data models. `https://github.com/smart-data-models/SmartCities`.

[5] Smart data models. `https://github.com/smart-data-models/Smart-Sensoring`.

[6] Fiware catalogue. `https://www.fiware.org/catalogue/`.

[7] Andrea Barone and Mirko Caforio. BDL_SAAS. `https://github.com/bandrea01/BDL_SAAS`, 2024.

[8] That open company. `https://thatopen.com/`.

[9] Jos de Jong. jsoneditor. `https://github.com/josdejong/jsoneditor.git`, 2011.