Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

IMPLEMENTAZIONE DI UNA THING DESCRIPTION DIRECTORY CON SUPPORTO PER RICERCA SEMANTICA

Relatore:
Chiar.mo Prof.
Davide Rossi

Presentata da:
Daniele Perrella

Sessione III
2022-2023

Indice

Capitolo 1

Introduzione

Nello scenario digitale odierno, l'interconnessione dei dispositivi che ci circondano, nota come Internet of Things (IoT), sta plasmando in modo rivoluzionario il modo in cui noi interagiamo con il mondo circostante. Attraverso l'uso di sensori intelligenti e connessioni ubiquitarie, l'IoT ha aperto le porte a scenari futuristici in cui gli oggetti prendono vita propria attraverso la raccolta e condivisione di dati in tempio reale.
Da questo contesto dinamico emerge il "Web of Things" (WoT)[1], un concetto avanzato che va oltre la semplice connettività, mirando a creare un ambiente dove le cose fisiche sono integrate in modo semantico nel vasto World Wide Web. Il Web of Things non è solo una rete di dispositivi intelligenti, ma piuttosto un ecosistema in cui gli oggetti interagiscono tra loro e con gli utenti in modo intelligente e contestualizzato.
L'integrazione di tecnologie semantiche in questo contesto mira quindi a migliorare la comprensione e l'interpretazione dei dati generati, aprendo nuove prospettive per applicazioni e servizi avanzati. Uno dei motivi principali per lo sviluppo del WoT è stato il riconoscimento della necessità di creare un framework che consentisse ai dispositivi IoT di comunicare in modo più significativo ed intelligente. Ciò include l'abilità di descrivere le funzionalità di un dispositivo in modo semantico, rendendo possibile l'interoperabilità tra dispositivi di diversi produttori e l'integrazione con servizi Web.
Organizzazioni come il W3C (World Wide Web Consortium)[2] hanno ricoperto un ruolo significativo nello sviluppo delle specifiche del WoT. Il W3C ha lavorato per definire standard aperti e protocolli che consentono la descrizione semantica delle cose fisiche e la loro interazione attraverso il Web. Grazie a tali sforzi collaborativi, hanno contribuito a plasmare il concetto di WoT, e quindi a fornire le fondamenta per lo sviluppo di applicativi e servizi basati su tale concetto.
In questo contesto, emergono applicativi che abbracciano alcune potenzialità del Web of Things, come Tiny-IoT[3], WotHive[4] e Zion[5], i quali saranno presentati nel capitolo 3. Questi progetti hanno svolto un ruolo cruciale nell'esplorazione iniziale del panorama WoT, fungendo da ispirazione per la definizione di tecnologie standard utilizzate. Sono stati quindi presi come punti di riferimento, fornendo informazioni riguardanti le

tecnologie attualmente in uso e le strategie di implementazione. Grazie a queste analisi preliminari è stato possibile stabilire l'approccio da utilizzare per lo sviluppo del progetto, che risulta essere un approccio sperimentale unico ed alternativo, attentamente progettato per rispondere alle specifiche esigenze del progetto. Questo approccio, sebbene differente dai progetti citati, rimane saldamente ancorato agli standard stabiliti dal W3C.
L'obiettivo principale del progetto consiste nella progettazione ed implementazione di una Things Directory conforme alle specifiche del WoT definite dal W3C, con particolare attenzione alle funzionalità di ricerca tramite query semantiche con linguaggio SPARQL. Il progetto mira a soddisfare le esigenze di un ambiente dinamico e distribuito, dove i devices possano interagire dinamicamente con la rete, promuovendo la flessibilità della rete stessa.
Per questo progetto sono stati stabiliti alcuni vincoli, così da diversificare il prodotto finale da quelli già disponibili, offrendo quindi una valida alternativa in base alle necessità. Tali vincoli saranno discussi capitolo 4.1.
Nel capitolo 4 sono esposte le diverse tecnologie prese in considerazione per la realizzazione del progetto, le quali mostrano argomenti a favore e contro al loro utilizzo. successivamente, nella sezione 4.2.4 vengono comparate a livello prestazionale. Al fine di soddisfare uno dei vincoli, il progetto presenta una differenza particolarmente interessante, relativa all'archiviazione e gestione dei dati, impiegando un approccio decisamente differente rispetto agli altri progetti. Questa comporta una miglioria importante per quanto riguarda le prestazioni dell'applicativo, e sarà discussa nel capitolo 5.2.
La progettazione e realizzazione del progetto sono discusse nel capitolo 5, illustrando le componenti del codice ed il loro comportamento, descrivendo approcci e design pattern utilizzati, costrutti e tecnologie utilizzate. In questo capitolo vengono quindi condivise le decisioni chiave prese per lo sviluppo del progetto per offrire una visione chiara del contributo all'ecosistema in evoluzione del Web of Things.

Capitolo 2

Web of Things (WoT) Discovery

Il Web of Things (WoT) del W3C è progettato per abilitare l'interoperabilità tra piattaforme IoT e domini applicativi diversi. Un elemento centrale per raggiungere questo obiettivo è la definizione e utilizzo di metadati che descrivono le interazioni messe a disposizione da un dispositivo o un servizio IoT sulla rete, ad un livello di astrazione appropriato. La specifica WoT Thing Description 2.1 soddisfa questo obiettivo, delineando in modo chiaro le caratteristiche delle Things.
Tuttavia, per utilizzare un Thing, è necessario prima ottenere la sua descrizione, e qui entra in gioco il processo di WoT Discovery. Questo processo, descritto nel draft W3C[1], si occupa di risolvere questo problema fondamentale. Il processo deve inoltre integrarsi con i meccanismi di discovery esistenti, garantire la sicurezza, proteggere le informazioni private e gestire efficientemente gli aggiornamenti delle Thing Descriptions, considerando la natura dinamica e diversificata dell'ecosistema IoT.

Il processo di WoT Discovery si suddivide in due fasi, *Introduzione* ed *Esplorazione*. La fase di *Introduzione* sfrutta meccanismi di discovery ma non espone direttamente i metadati; essi vengono utilizzati semplicemente per scoprire i servizi di *Esplorazione*, che forniscono i metadati solo dopo un'autenticazione e autorizzazione sicure.

Il processo di *Esportazione* è a sua volta diviso in due servizi:

- Distribuzione di Thing Individuali: Tramite un servizio web regolare
- Servizio di Thing Directory: che consente la ricerca di collezioni di Thing Descriptions.

L'interesse dello studio corrente è posto sul servizio **Thing Directory**

2.1 Thing Description

Le Thing Description costituiscono il fondamento informativo del Web of Things, fornendo una rappresentazione semantica e dettagliata delle entità connesse. In questa sezione esploreremo le caratteristiche chiave e l'importanza delle Thing Description nel contesto del WoT.

2.1.1 Definizione e Struttura delle Thing Description

Le Thing Description (TD) sono documenti standardizzati che descrivono in modo dettagliato le caratteristiche, le funzionalità e le interfacce di un'entità del WoT. Ogni TD è strutturata seguendo le specifiche stabilite dal W3C e contiene informazioni essenziali per consentire una comprensione approfondita e una corretta interazione con l'entità descritta.

La struttura di base di un Thing Description include:

- *'@context'*: Definisce il contesto semantico della descrizione, specificando il significato dei termini utilizzate nel documento. Questo elemento facilità l'interpretazione univoca delle informazioni contenute nella TD.
- *'@type'*: Indica il tipo di entità descritta, ad esempio, "Thing", "Property", "Action" o "Event". Questo elemento fornisce informazioni sulla natura dell'oggetto descritto.
- *'title' e 'description'*: Forniscono un titolo ed una descrizione testuale che aiutano a identificare e comprendere l'entità. Queste informazioni sono utili per gli sviluppatori e gli utenti finali.
- *'@id'*: Indica l'identificativo univoco dell'entità. Questo permette di effettuare operazioni dirette su specifiche Thing Description, riuscendo ad identificarle all'interno del sistema.

Per supportare tale struttura, non basta però il formato standard utilizzato da applicazioni **RESTful** ovvero **JSON**, bensì, bisogna adottare un nuovo formato che includa alcuni nuovi funzionamenti: **JSON-LD**

2.1.2 Formato JSON-LD

JSON-LD (JSON Linked Data)[30] è una estensione del formato **JSON** progettata per consentire la serializzazione dei *Linked Data* all'interno del classico formato **JSON**, creando così una struttura di dati interconnessi ed interpretabile dalle macchine.

Caratteristiche Principali di JSON-LD

Questo formato presenta le seguenti caratteristiche principali:

- **Integrazione Aggiornabile**: **JSON-LD** è stato progettato per integrarsi facilmente nei sistemi che già fanno uso del formato **JSON**, offrendo un percorso di aggiornamento fluido da **JSON** verso **JSON-LD**. Questo permette un'adozione agevolata dei *Linked Data* negli ambienti di programmazione web-based.
- **Strumento per la Creazione di Linked Data Networks**: **JSON-LD** facilita la creazione di reti di dati interoperabili attraverso il collegamento tra diversi documenti e siti web, utilizzando i principi dei *Linked Data*. Le applicazioni possono difatti navigare tra diverse risorse mediante i collegamenti incorporati.
- **Identificazione Universale**: **JSON-LD** introduce un particolare meccanismo di identificazione universale tramite gli **Internationalized Resource Identifiers** (**IRIs**), permettendo il riferimento univoco di oggetti JSON.
- **Context per Disambiguazione delle Chiavi**: Per risolvere problemi di ambiguità tra chiavi condivise tra diversi documenti **JSON**, **JSON-LD** utilizza una proprietà denominata *"@context"*, necessaria per mappare le chiavi in *IRI*, garantendone quindi una chiarezza semantica.
- **Interoperabilità con JSON**: **JSON-LD** può essere utilizzato direttamente come un classico **JSON** privo di funzionalità **RDF** (Resource Description Framework), offrendo così l'interoperabilità senza soluzione di continuità con l'ampio ecosistema di parser e librerie **JSON** esistenti.
- **Annotazioni Linguistiche**: **JSON-LD** permette di associare alle proprietà testuali la lingua di appartenenza, permettendo quindi una descrizione più ricca dei dati.

2.1.3 Importanza delle Thing Description

Le Thing Description svolgono un ruolo cruciale nel contesto del WoT per diversi motivi:

- *Interoperabilità*: Le TD forniscono uno standard comune per descrivere entità, consentendo quindi l'interoperabilità tra dispositivi e servizi di diversi produttori. Ciò facilita la creazione di ecosistemi WoT coesi e collaborativi.
- *Comprensione Semantica*: Grazie all'utilizzo di un contesto semantico definito, le TD consentono una comprensione più profonda e contestualizzata dell'entità. Gli sviluppatori possono interpretare e utilizzare le informazioni in modo più preciso.

- *Semplicità di Integrazione*: Le informazioni dettagliate nelle TD semplificano l'integrazione di nuove entità all'interno di applicazioni esistenti. Gli sviluppatori ed i sistemi possono comprendere rapidamente le funzionalità offerte senza la necessità di consultare documentazioni esterne.

- *Adattabilità ed Estensibilità*: La struttura modulare delle TD consente di estendere ed adattare le descrizioni in base alle esigenze specifiche del sistema. Questa flessibilità favorisce lo sviluppo di soluzioni su misura.

In conclusione, le Thing Description rappresentano una componente chiave nell'ecosistema del Web of Things, facilitano la standardizzazione, la comprensione e l'interoperabilità tra le entità connesse. La loro adozione è fondamentale per la realizzazione di un WoT dinamico e collaborativo.

2.2 Thing Directory

La Thing Directory è l'elemento cardine del Web of Things (WoT), svolgendo un ruolo centrale nella creazione di un ambiente coeso e collaborativo. In questa sezione, esploreremo approfonditamente il ruolo e le caratteristiche chiave della Thing Directory, concentrandoci su funzionalità avanzate come le notifiche degli eventi e le robuste opzioni di ricerca sintattica e semantica tramite i particolari linguaggi di querying JSONPath[12], XPath[11] e SPARQL[10].

2.2.1 Funzionalità Offerte

La Thing Directory funge da archivio centrale in cui le entità WoT, fisiche o virtuali, sono registrate ed identificate in modo semantico. Tra le funzionalità fondamentali spiccano:

- **Thing Description**: Ogni entità viene descritta dettagliatamente attraverso standard WoT, definendo chiaramente le sue caratteristiche e funzionalità. Questo approccio semantico garantisce una comprensione approfondita delle capacità di ciascun dispositivo o servizio.

- **Operazioni CRUD**: La Thing Directory svolge un ruolo dinamico nel ciclo di vita delle entità WoT, fornendo un'interfaccia robusta per la gestione delle operazioni CRUD. Questo significa che gli utenti e gli sviluppatori possono interagire con la Thing Directory per:

  - **Creare**: Aggiungere nuove entità WoT al registro. Questo processo coinvolge la definizione dettagliata di Thing Description, garantendo una registrazione accurata e semantica all'interno dell'ecosistema WoT.

- **Leggere**: Recuperare informazioni dettagliate sulle entità esistenti, comprese le loro caratteristiche, le funzionalità e le relazioni con altre entità. La Thing Directory agisce come una fonte centralizzata di conoscenza, semplificando la consultazione delle informazioni di interesse.
- **Aggiornare**: Modificare le informazioni esistenti, mantenendo la coerenza e l'accuratezza delle Thing Description. Gli utenti possono apportare modifiche alle entità WoT per riflettere cambiamenti nelle loro caratteristiche o funzionalità nel tempo.
- **Eliminare**: Rimuovere entità non più rilevanti o necessarie dall'ecosistema WoT. Questo processo è fondamentale per mantenere un registro pulito e aggiornato, riflettendo con precisione lo stato attuale delle entità connesse.

L'implementazione di queste operazioni CRUD nella Thing Directory garantisce una gestione flessibile e dinamica delle entità WoT, consentendo agli utenti di adattare ed evolvere l'ecosistema in modo coerente con le mutevoli esigenze dell'applicazione e dell'ambiente circostante.

- **Ricerca e Scoperta Avanzate** *(Opzionale)*: La Thing Description Directory supporta tre API di ricerca: ricerca **sintattica** con **JSONPath**[12], ricerca sintattica con **XPath**[11] e ricerca semantica con **SPARQL**[10]. La Directory ha la facoltà di implementare la ricerca semantica con SPARQL, mentre JSONPath e XPath sono opzionali e soggette a modifiche future. È altamente raccomandato dalle specifiche che le directory implementino almeno un'API di ricerca per servire efficientemente le Thing Description basate su query specifiche del client.
  - **Ricerca Flessibile e Sintattica tramite JSONPath**: Consente di definire query flessibili e mirate utilizzando espressioni JSONPath. Ciò significa che è possibile recuperare informazioni specifiche all'interno delle Thing Definition in modo rapido e preciso.
  - **Ricerca Flessibile e Sintattica tramite XPath**: Offrono un'alternativa potente per coloro che sono più familiari con le espressioni XPath. La possibilità di eseguire query basate sulla struttura del documento XML può semplificare la ricerca e il filtraggio dei dati.
  - **Ricerca Semantica e Interrogazioni Complesse con SPARQL**: L'implementazione delle API di ricerca semantica con **SPARQL** introduce la capacità di eseguire interrogazioni complesse che coinvolgono relazioni semantiche tra le Thing Description. Gli sviluppatori possono sfruttare la potenza del linguaggio SPARQL per ottenere risultati specifici basati su connessioni semantiche.
    La possibilità di eseguire interrogazioni **SELECT**, **ASK**, **CONSTRUCT**

o **DESCRIBE** apre ulteriori opzioni di ricerca, consentendo agli utenti di adattare le loro richieste in base alle esigenze specifiche.

- **Supporto per la Federated Query via SPARQL**: Nel caso in cui la Thing Description Directory implementi il supporto per le Federated Query, si aprono nuove prospettive di interoperabilità. Si possono eseguire query distribuite su più directory, facilitando la ricerca di informazioni in ambienti distribuiti e complessi.
- **Possibilità di Estensione e Personalizzazione**: La modularità delle API di ricerca consente di estendere e personalizzare le funzionalità in base alle esigenze specifiche del progetto. L'introduzione di nuove espressioni di ricerca o l'estensione delle query esistenti può gestire essere gestita in modo agevole.

In sintesi, l'implementazione di queste API offre un'ampia gamma di possibilità, dalla ricerca sintattica rapida e mirata alla navigazione semantica complessa. Gli sviluppatori possono sfruttare tali opportunità per creare applicazioni più intelligenti ed adattabili, migliorando l'esperienza complessiva degli utenti nell'utilizzo delle Thing Description.

- **Notifiche degli Eventi** *(Opzionale)*: La Thing Directory supporta la notificazione degli eventi attraverso l'implementazione dell'API di notifica. Questa funzionalità è fondamentale per mantenere gli utenti e gli sviluppatori informati sugli aggiornamenti significativi all'interno dell'ecosistema WoT.

  L'API di notifica segue le specifiche del Server-Sent Events (SSE)[17], consentendo al server di inviare eventi ai client gli aggiornamenti. Gli eventi catturando le fasi del ciclo di vita delle Thing Description all'interno della directory e possono essere dei seguenti tipi:

  - **thing_created**: Quando viene inserita (o creata) una nuova Thing.
  - **thing_updated**: Quando viene modificata una Thing già presente all'interno del sistema.
  - **thing_deleted**: Quando viene rimossa una Thing dal sistema

  La gestione degli eventi offre le seguenti caratteristiche:

  - **Eventi Attribuiti al Ciclo di Vita**: Gli eventi notificati dal server sono strettamente legati al ciclo di vita delle Thing Description, includendo tipi come *thing_created*, *thing_updated* e *thing_deleted*. Questo permette di ricevere notifiche specifiche sulle modifiche rilevanti alle entità WoT.
  - **Filtri degli Eventi**: L'API consente la filtrazione degli eventi lato server, riducendo il consumo di risorse consegnando solo gli eventi richiesti dai client.

Ad esempio, i client possono sottoscriversi per ricevere solo gli eventi relativi alla creazione di nuove Thing Description, ottimizzando la rilevanza delle notifiche.

- **Persistenza degli Eventi**: La persistenza degli eventi è implementata secondo le specifiche SSE, consentendo ai client di riconnettersi e ricevere tutti gli eventi persi durante la disconnessione. Il server assegna un ID ad ogni evento e risponde alle richieste di riconnessione fornendo tutti gli eventi mancanti.
- **Dati degli Eventi**: Gli eventi notificati contengono la serializzazione JSON dell'oggetto evento. L'oggetto evento è una Thing Description parziale o completa, a seconda della richiesta. All'interno dell'evento è incluso *almeno* l'identificatore della TD creata, aggiornata o eliminata.

L'implementazione dell'API di notifica nella Thing Directory garantisce una comunicazione reattiva e tempestiva sugli eventi rilevanti, fornendo un meccanismo essenziale per la sincronizzazione e l'aggiornamento continuo delle Thing Description nell'ecosistema WoT.

Nel contesto della Thing Directory, queste funzionalità consentono una gestione completa delle entità WoT, facilitando la ricerca, la scoperta e la manipolazione di dispositivi e servizi all'interno dell'ecosistema Web of Things.

2.2.2 Innovazioni Tecnologiche: RDF & Triple Store

Nell'affrontare l'implementazione di una Thing Directory, ci si trova di fronte ad una sinergia unica tra tecnologie avanzate, le quali contribuiscono significativamente alla gestione ed organizzazione delle informazioni semanticamente arricchite. Tra le tecnologie chiave delle Thing Directory, per quanto riguarda le features semantiche, è necessario evidenziare **RDF** (Resource Description Framework) e **Triple Store**. Questi, sono strumenti potenti e flessibili, che ci permettono di plasmare l'ecosistema dinamico ed interconnesso di entità smart.

RDF: Le Fondamenta Semantiche dell'Ecosistema

Il **Resource Description Framework** (**RDF**)[8] è la spina dorsale del web semantico. Si tratta di un linguaggio che consente la descrizione delle risorse e delle relazioni tra di esse in maniera strutturata e ricca di significato. All'interno di una Thing Directory, **RDF** assume un ruolo cruciale nell'aprire le porte ad una rappresentazione semantica avanzata delle informazioni. Questo framework consente la modellazione delle **things** in modo flessibile, indicando chiaramente i concetti e le connessioni che li coinvolgono. Grazie a **RDF**, la Thing Directory diviene un sistema in grado di interpretare ed organizzare le entità in un contesto comprensibile, sfruttando l'utilizzo di una sintassi

espressiva per la definizione di relazioni ed attributi. Il modello dei dati del formato **RDF** è strutturato intorno a tre elementi chiave: il **soggetto** (**subject**), il **predicato** (**predicate**) e l'**oggetto** (**object**). Questi tre componenti fondamentali costituiscono ciò che è noto come *"RDF Triple"* e rappresentano l'essenza della rappresentazione semantica delle informazioni.
Nel dettaglio:

- **Soggetto (Subject)**: Il **subject** in **RDF** rappresenta l'entità principale (o la risorsa) su cui si stanno fornendo le informazioni. Può essere quindi qualsiasi cosa, da oggetti fisici a concetti astratti.
- **Predicato (Predicate)**: Il **predicate** in **RDF** indica la relazione o l'attributo che collega un oggetto al relativo soggetto. Questo fornisce il contesto semantico per la relazione tra le varie entità.
- **Oggetto (Object)**: L'**object** in **RDF** rappresenta il valore o la risorsa associata ad un dato predicato nel contesto della **RDF Triple**. Questo può quindi avere un valore letterale, un valore numerico, un oggetto complesso o addirittura un'altra risorsa.

Le **RDF Triple** lavorano con sinergia, modellando le relazioni semantiche tra le entità. Inoltre, le **RDF Triple** possono essere concatenate per arrivare a formare grafi complessi, creando rappresentazioni interconnesse e semantiche delle informazioni.
L'utilizzo di **subject**, **predicate** ed **object** conferiscono significato semantico alle informazioni. Anziché rappresentare i dati in modo statico, **RDF** permette la modellazione del significato delle relazioni tra **things**, facilitando l'interpretazione delle informazioni sia da parte di macchine che di esseri umani.

Di seguito, un esempio del codice[34] dietro una RDF Triple e la relativa visualizzazione sotto forma di grafo:

In sintesi, questa combinazione di **subject**, **predicate** ed **object** in **RDF** costituisce la base per la creazione di un web semantico, dove le informazioni sono rappresentate in un formato con una determinata struttura e significato, promuovendo la comprensione e interoperabilità avanzate delle risorse digitali (e di conseguenza, della loro controparte fisica).

Triple Store: L'Archivio delle Triple

Un elemento cruciale per quanto riguarda la gestione dei dati semantici, è rappresentato dai **Triple Stores**. Questi sistemi specializzati forniscono un ambiente robusto e scalabile per la memorizzazione, ricerca e interrogazione di informazioni semantiche modellate attraverso il framework **RDF**.

```
:John a :Man ;
    :name "John" ;
    :hasSpouse :Mary .
:Mary a :Woman ;
    :name "Mary" ;
    :hasSpouse :John .
:John_jr a :Man ;
    :name "John Jr." ;
    :hasParent :John, :Mary .
:Time_Span a owl:Class .
:event a :Activity ;
    :has_time_span [
        a :Time_Span ;
        :at_some_time_within_date "2018-01-12"^^xsd:date
        ] .
:u129u-klejkajo-2309124u-sajfl a :Person ;
    :name "John Doe" .
```

Listing 1: RDF Triple[34]

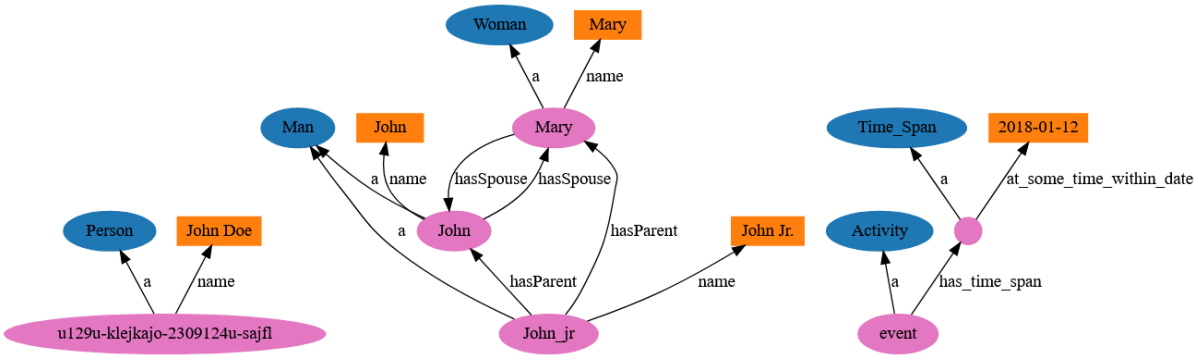

Figura 2.1: OWL Visualization [34]

Un **Triple Store** è un tipo di database progettato appositamente per l'archiviazione e il recupero di **RDF Triple**. Il **Triple Store** svolge quindi un ruolo chiave nella creazione del web semantico, consentendo il mantenimento delle informazioni connesse. Di seguito sono illustrate alcune caratteristiche chiave dei **Triple Store**:

- **Struttura di un Triple Store**: La funzionalità principale di un **Triple Store** è la sua capacità di organizzare e strutturare le **RDF Triple**. Comunemente, i **Triple Store** utilizzano una struttura a grafo, dove le risorse sono nodi, e le relazioni tra di esse sono rappresentate dagli archi tra i nodi coinvolti. Questa struttura consente una rapida navigazione ed il recupero di informazioni attraverso le connessioni.

- **Archiviazione e Recupero**: I **Triple Store** sono ottimizzati per l'archiviazione e un recupero efficiente di enormi quantità di **RDF Triple**. Possono sfruttare diverse tecniche di indicizzazione e compressione per garantire performance elevate durante le interrogazioni, specialmente quando il numero di triple aumenta considerevolmente.

- **Interrogazioni SPARQL**: Un aspetto fondamentale dei **Triple Store** è la possibilità di eseguire interrogazioni tramite il linguaggio di query **SPARQL**. Questo permette l'estrazione di informazioni specifiche, filtrando i dati e scoprendo relazioni tra risorse. i **Triple Store** forniscono un'implementazione efficiente di **SPARQL** per facilitare l'accesso a informazioni semantiche in modo potente e flessibile.

- **Gestione Grafi RDF**: I **Triple Store** gestiscono grafi RDF complessi, consentendo la creazione e gestione di strutture dati intricate e interconnesse. Questa gestione di grafi è essenziale per riflettere accuratamente le relazioni tra le risorse, in modo tale da creare un contesto semantico ricco.

- **Integrazione con Tecnologie per Web Semantico**: I **Triple Store** sono spesso progettati per essere integrati con le tecnologie avanzate per il Semantic Web, come ad esempio **RDFS** (Resource Description Framework Schema) e **OWL** (Web Ontology Language). Queste tecnologie integrano ulteriore complessità alle rappresentazioni, consentendo la modellazione di gerarchie, la definizione di vincoli e di fornire una semantica più sofisticata.

La visualizzazione del codice RDF mostrato in precedenza2.1 viene riassunto all'interno di un Triple Store utilizzando esclusivamente le triple *subject*, *predicate* ed *object*, e possiamo visualizzarlo nel seguente modo:

Tabella 2.1: RDF Data Table

Subject	**Predicate**	**Object**
:John	rdf:type	:Man
:John	:name	"John"
:John	:hasSpouse	:Mary
:Mary	rdf:type	:Woman
:Mary	:name	"Mary"
:Mary	:hasSpouse	:John
:John_jr	rdf:type	:Man
:John_jr	:name	"John Jr."
:John_jr	:hasParent	:John, :Mary
:event	rdf:type	:Activity
:event	:has_time_span	:_anon1
:_anon1	rdf:type	:Time_Span
:_anon1	:at_some_time_within_date	"2018-01-12"xsd:date
:Time_Span	rdf:type	owl:Class
:u129u-klejkajo-2309124u-sajfl	rdf:type	:Person
:u129u-klejkajo-2309124u-sajfl	:name	"John Doe"

In sintesi, un **Triple Store** è il cuore delle applicazioni basate su **RDF**, fornendo una piattaforma dedicata per la gestione avanzata dei dati semantici, facilitando la creazione di un ambiente web intelligente e connesso.
Questa architettura offre un vantaggio distintivo, permettendo una navigazione efficiente e veloce dei grafi **RDF**. Per le **Thing Directory** questo significa accesso accurato e rapido alle informazioni sulle things, facilitandone la ricerca, le query su di esse e l'aggiornamento delle entità presenti nel sistema.

Capitolo 3

Related Works

Questo capitolo rappresenta una panoramica dei progetti correlati che si allineano alle specifiche delineate dal W3C per le Thing Description Directory. Tutti i progetti considerati sono di natura Open Source, con le relative sorgenti accessibili sulla piattaforma GitHub[6]. L'analisi di queste iniziative fornisce un contesto essenziale per comprendere le diverse implementazioni e strategie adottate nel campo delle Thing Description Directory.

L'obiettivo principale di esplorare i lavori correlati è quello di evidenziare le sfide comuni affrontate nell'ambito della gestione delle Thing Description, nonché di identificare le soluzioni innovative proposte da altri progetti. Attraverso questa analisi comparativa, sarà possibile trarre ispirazione dalle esperienze altrui, e valutare le best practices adottate dalla comunità Open Source.
Ciascun progetto presentato sarà esaminato sotto diversi aspetti, tra cui la sua architettura generale, le tecnologie utilizzate, le funzionalità offerte e le scelte di progettazione. questo permetterà di ottenere una visione completa delle varie implementazioni, e fornirà un contesto significativo per la progettazione e lo sviluppo della Thing Description Directory prevista in questo lavoro.

3.1 Tiny-Iot

Tiny-IoT[3] è un'implementazione di una Thing Directory realizzata con il linguaggio Go[7]. Il progetto si concentra sulla creazione di un registro di Things (thing Directory) interoperabile, in cui le things possono essere registrate e identificate, offrendo supporto ad alcuni metodi di ricerca tramite querying con linguaggi appositi.

3.1.1 Caratteristiche Principali

Di seguito sono riportate le caratteristiche principali che hanno reso il progetto interessante per lo studio:

- **Progettato per la Scalabilità**: Il progetto è progettato per essere scalabile, consentendo una gestione efficiente di un numero crescente di dispositivi e oggetti connessi grazie all'implementazione con linguaggio Go[7].
- **Copertura delle features elencate dal W3C**: Supporta molte delle funzionalità non obbligatorie richieste dal W3C, come la ricerca di Things tramite Query Languages:
  - JSONPath[12].
  - XPath[11].

  Non offre però supporto al linguaggio di query SPARQL[10].
- **Deployment**: La directory è disponibile come Docker Image[13] e Binary Distribution.
- **Risorse molto limitate**: L'implementazione in Go consente un'esecuzione molto leggera a livello prestazionale.
- **Archiviazione**: L'archiviazione dei dati avviene tramite la libreria LevelDB[14], una libreria di database open-source offerta da Google[15]. Tale libreria è una soluzione di memorizzazione key-value, offrendo un'implementazione di un database incorporato, leggero e performante, oltre che essere persistente.
- **Prestazioni in fase di lookup**: Nonostante non siano supportate funzionalità semantiche, la soluzione di archiviazione utilizzata garantisce elevate prestazioni in fase di lookup, potendo effettuare la ricerca su un database di tipo key-value.
- **Performance generali**: Il linguaggio Go è noto grazie alle sue prestazioni elevate. La sua gestione efficiente delle routine, i meccanismi di garbage collection leggeri e l'approccio alla concorrenza, facilitano la gestione dei carichi di lavoro intensi, il che può essere cruciale per una Thing Directory che gestisce un gran numero di entità.

3.1.2 Svantaggi

- **Mancato supporto per SPARQL**: La mancanza del supporto alle funzionalità di ricerca semantica tramite SPARQL penalizza il progetto, limitandone molto il potenziale.

Pur essendo un'applicazione interessante e promettente, presenta un limite significativo in termini di ricerca semantica. La limitazione impedisce al progetto di essere considerato un esempio completo e versatile. Pertanto, mentre TinyIoT offre approcci validi ed interessanti per determinate esigenze, è importante considerare alternative che possano soddisfare pienamente i requisiti di ricerca semantica nelle implementazioni di Thing Directory.

3.2 WotHive

WoTHive è un'implementazione di Thing Directory, che include anche gli aspetti semantici, supportando il linguaggio di querying SPARQL[10]. Questo lo rende l'implementazione più completa presa in considerazione, ma presenta alcuni svantaggi a livello prestazionale.

3.2.1 Caratteristiche Principali

Di seguito sono riportate le caratteristiche principali che hanno reso il progetto interessante per lo studio:

- **Configurazione**: Permette di configurare l'applicativo, modificando determinate impostazioni prima dell'esecuzione del sistema.
- **Copertura delle features elencate dal W3C**: Supporta le funzionalità non obbligatorie richieste dal W3C, inclusa la ricerca di Things tramite linguaggi di query:
  - JSONPath[12].
  - SPARQL[10].
- **Archiviazione**: L'archiviazione avviene su un triple-store, così da consentire l'utilizzo del linguaggio di query semantiche SPARQL. Questo triple store non è però incorporato nell'applicativo stesso, ma richiede l'esistenza di un'unità di archiviazione persistente (di tipo triple store) esterna. Questa scelta potrebbe risultare efficace per quanto riguarda la personalizzazione del sistema, permettendo all'utente di selezionare il triple store di propria preferenza purché sia compatibile.

- **Docker**: WotHive offre un'implementazione accessibile e immediata grazie all'uso di **docker** [13]. Con un unico comando gli utenti possono avviare l'intero ambiente WoTHive, compreso un triple store, e tutti i servizi associati. Questo favorisce l'accessibilità del progetto, ma comporta l'installazione necessaria di altri applicativi accessori.

- **Implementazione**: L'applicativo è realizzato con il linguaggio Java[16], il che permette di utilizzare librerie già consolidate, garantendo un ecosistema sicuro e stabile.

3.2.2 Svantaggi

Come ogni progetto, l'applicativo presenta alcune criticità che meritano attenzione. Di seguito saranno esaminati in dettagli gli aspetti che richiedono considerazione

- **Applicativi Terzi**: L'integrazione di applicativi terzi introduce un livello di complessità aggiuntiva al sistema. La cooperazione tra diverse componenti potrebbe generare punti di potenziale vulnerabilità. Un malfunzionamento in una di queste componenti potrebbe influire sull'intero applicativo, evidenziando la necessità di una gestione attenta.

- **Operazioni di lookup**: L'esecuzione di numerose richieste di lookup su un triple store potrebbe comportare sfide prestazionali, specialmente in assenza di meccanismi di caching efficienti. Ciò può tradursi in prestazioni ridotte per operazioni di lookup apparentemente semplici, come la ricerca per ID.

- **Implementazione**: L'utilizzo del linguaggio Java, sebbene vantaggioso in diversi aspetti, presenta sfide nella gestione dell'asincronia. La complessità associata alla gestione degli aspetti asincroni potrebbe richiedere un'attenzione particolare. Altri linguaggi che sfruttano costrutti come *coroutines* o *promises* possono offrire soluzioni più fluenti in contesti ad attività asincrone intense, come potrebbe essere una Thing Directory.

Questa sezione delinea gli aspetti critici che richiedono una valutazione approfondita durante lo sviluppo e l'implementazione. La consapevolezza di determinati svantaggi può rivelarsi fondamentale per una gestione efficace e per la ricerca di soluzioni che minimizzino gli impatti negativi.

3.3 Zion

Zion è un progetto attualmente in fase alpha, ma risulta interessante e promettente grazie all'approccio differente rispetto gli altri progetti presi in considerazione. Questo progetto

mostra infatti un potenziale significativo pur trovandosi ancora in uno stadio di sviluppo iniziale. Questa sezione fornisce una panoramica preliminare di Zion, evidenziando alcuni aspetti promettenti ed alcune sfide attuali.

3.3.1 Caratteristiche Principali

- **Implementazione**: Zion sfrutta TypeScript, un linguaggio che offre un supporto avanzato per la gestione delle attività asincrone. Questa scelta potrebbe semplificare notevolmente l'implementazione delle funzionalità asincrone richieste da una Thing Directory. Questo potrebbe tradursi in una maggiore efficienza nella gestione delle attività asincrone intense.

- **Ecologia JavaScript**: TypeScript si basa su JavaScript, il linguaggio di scripting più diffuso. Ciò significa che gli sviluppatori possono sfruttare le librerie e i framework esistenti nel vasto ecosistema di JavaScript, minimizzando il codice da implementare in prima persona.

3.3.2 Sfide Attuali

Nonostante ciò presenta alcune sfide o limitazioni che lo rendono un esempio di approccio meno appetibile.

- **Limitazioni Funzionali**: Zion è attualmente in uno stato di alpha, indicando che sono presenti limitazioni funzionali e che l'implementazione non è ancora completa. L'uso in ambienti di produzione potrebbe richiederne ulteriori sviluppi e miglioramenti.

- **Dipendenze da NPM**: Un aspetto critico da considerare è la dipendenza da librerie NPM, che potrebbe comportare una notevole quantità di librerie da scaricare durante l'installazione. Questa caratteristica potrebbe richiedere maggiore spazio di allocazione del necessario.

Zion, nonostante sia in una fase iniziale, mostra comunque potenzialità, soprattutto per quanto riguarda la gestione asincrona. Tuttavia, è importante valutare attentamente le limitazioni attuali, specialmente nell'ambito dell'utilizzo di librerie NPM, che lo rendono una soluzione poco appetibile per sistemi con risorse limitate.

Capitolo 4

Selezione delle Tecnologie

Come discusso nel capitolo 3, nel processo di sviluppo di qualsiasi progetto, la selezione delle tecnologie gioca un ruolo cruciale nel determinare il successo e l'efficienza dell'implementazione. Questo capitolo si concentra sulla fondamentale decisione di scelta dei vincoli imposti per il progetto, la selezione del linguaggio di programmazione e delle librerie da utilizzare nel contesto dello sviluppo di una Thing Directory. La scelta di queste tecnologie non solo influenzerà la struttura e le prestazioni del sistema, ma avrà anche un impatto significativo sulla manutenibilità, sull'estensibilità e sulla facilità d'uso del prodotto finale. Esploriamo attentamente le motivazioni dietro la selezione delle tecnologie adottate, analizzando le caratteristiche distintive di ciascuna opzione e valutando come possano contribuire in modo sinergico al successo complessivo del progetto.

4.1 I Vincoli

Nel contesto di questo progetto, la scelta delle tecnologie non è solo una questione di efficienza e funzionalità, ma anche di adattamento a vincoli specifici. Sono stati quindi identificati d imposti dei vincoli fondamentali, tra i quali si troverà un equilibrio, delineando le scelte tecnologiche ed architetturali.

- **Minor Numero di Dipendenze Possibile**: La volontà di mantenere il sistema snello e mantenibile comporta a ridurre al minimo il numero di dipendenze esterne. Ogni libreria o framework proposto sarà attentamente valutato, assicurandosi che la sua inclusione sia fondamentale per il successo del progetto. Inoltre, questo aiuterà a mantenere il pacchetto finale di dimensioni inferiori.
- **Nessuna Installazione Aggiuntiva di Strumenti o Environment**: L'installazione di componenti necessarie è limitata esclusivamente al framework di esecuzione per supportare il linguaggio in questione utilizzato per la realizzazione del progetto, così da permettere l'esecuzione del programma.

- **Triple Store Embedded per Escludere Componenti Esterne**: Per garantire un'implementazione autonoma e minimizzare le dipendenze esterne, si richiede l'utilizzo di un embedded triple store direttamente all'interno dell'applicazione. Ciò elimina la necessità di componenti aggiuntive o servizi esterni per la gestione dei dati RDF, semplificando ulteriormente l'architettura e ottimizzando le prestazioni. Questo vincolo assicura che la Thing Directory sia in grado di gestire in modo indipendente le operazioni legate ai dati RDF senza richiedere l'uso di triple store esterni, contribuendo così alla coerenza e all'autonomia dell'applicazione.

- **Un Eseguibile, Ready-to-Use**: La Thing Directory sarà distribuita come un singolo file eseguibile, pronto per l'uso e senza la necessità di complessi processi di installazione o configurazione. Questo includerà anche le librerie necessarie, senza la necessità di doverle installare manualmente. L'obiettivo è massimizzare la semplicità di distribuzione, favorendo l'integrazione su diversi dispositivi con un'unica soluzione pronta all'uso.

- **Ideale per Sistemi con Hardware Limitato**: La Thing Directory è progettata per funzionare su dispositivi con risorse limitate, come dispositivi IoT o hardware embedded. Questo vincolo guiderà le nostre scelte tecnologiche per garantire ottimizzazione delle prestazioni e adattabilità a contesti con risorse limitate. Adattare la Thing Directory a risorse limitate è cruciale per consentire un'ampia adozione su dispositivi embedded, garantendo un funzionamento efficiente ed affidabile.

Questi vincoli delineano la cornice delle decisioni di progettazione, assicurando che la Thing Directory si sviluppi in modo armonioso e risponda in modo efficace alle esigenze specifiche del progetto.
La progettazione di sistemi complessi spesso implica la necessità di trovare un equilibrio tra vincoli contrastanti. In particolare, nel contesto di una Thing Directory, bisogna riconoscere che il soddisfacimento completo di tutti i vincoli potrebbe rappresentare una sfida importante. Pertanto, è fondamentale adottare un approccio bilanciato, dove alcuni vincoli potrebbero essere parzialmente sacrificati per ottenere vantaggi significativi in altri aspetti del progetto stesso, accettando il compromesso.

4.2 Linguaggio

Per la progettazione della Thing Directory, la scelta del linguaggio di programmazione svolge un ruolo cruciale nella definizione dell'architettura e delle prestazioni complessive del sistema. Tra le opzioni considerate (**C#**, **Java** e **Kotlin**) l'obiettivo è individuare il linguaggio che meglio si adatta alle esigenze specifiche del progetto, offrendo stabilità, prestazioni e innovazione.
I linguaggi **C#** e **Java** sono noti per la loro stabilità ed affidabilità. Entrambi sono

linguaggi consolidati, utilizzati ampiamente in settori critici ed aziendali. La robustezza di questi linguaggi fornisce una base solida per lo sviluppo di un'applicazione che richiede consistenza e prestazioni affidabili, e saranno presi in considerazione per questo.
A questi due linguaggi viene affiancato un terzo, *Kotlin*, che si distingue come un'alternativa innovativa a *Java*, sfruttando le stesse basi.

La scelta del linguaggio di programmazione rappresenta un equilibrio delicato tra stabilità comprovata e l'opportunità di introdurre innovazioni che migliorino l'efficienza e la manutenibilità del codice. Nel capitolo seguente, saranno esaminate in dettaglio le caratteristiche dei linguaggi **C#**, **Java** e **Kotlin**, confrontando le loro capacità di soddisfare le esigenze specifiche della Thing Directory e delineando la strada per una scelta informata e strategica.

4.2.1 DotNet: C#

C# (*C-Sharp*) è un linguaggio di programmazione orientato ad oggetti sviluppato da Microsoft[21] nel 2000 come parte della piattaforma **.NET**[22]. Questo linguaggio è progettato per essere moderno, efficiente ed orientato agli oggetti, con una particolare enfasi sulla sicurezza e facilità d'uso.
Di seguito, le caratteristiche chiave di *C#* che lo rendono un candidato rilevante per la progettazione della Thing Directory.

Caratteristiche Principali

- **Orientamento agli Oggetti**: *C#* è un linguaggio fortemente orientato agli oggetti, consentendo agli sviluppatori di organizzare il codice intorno a concetti chiave come classi e oggetti. Inoltre sono supportati concetti fondamentali come incapsulamento, ereditarietà e polimorfismo. Questo paradigma risulta particolarmente adatto per modellare entità complesse coinvolte nella Thing Directory.

- **Gestione Automatica della Memoria**: *C#* utilizza un sistema di **garbage collection** per la gestione automatica della memoria, liberando gli oggetti non utilizzati, ottimizzando quindi l'utilizzo della memoria da parte dell'applicativo.

- **Sicurezza e Tipizzazione Statica**: *C#* è dotato di un sistema di tipizzazione statica, il che significa che gli errori relativi ai tipi vengono identificati durante la compilazione anziché durante il runtime. Questo aspetto contribuisce a creare codice più robusto e a prevenire potenziali errori critici.

- **Integrazione con *.NET***: *C#* è strettamente integrato con il framework **.NET**, offrendo l'accesso ad un vasto ecosistema di librerie e servizi. Questa integrazione semplifica lo sviluppo e fornisce soluzioni standardizzate per molte sfide comuni.

- **Asincronia e Await**: *C#* supporta **nativamente** la programmazione asincrona attraverso le parole chiave '*async*' e '*await*'. Questo risulta cruciale per la gestione di operazioni di **I/O** non bloccanti, migliorando notevolmente l'efficienza complessiva dell'applicazione.

- **LINQ (Language Integrated Query)**: **LINQ** è una caratteristica potente di *C#* che consente di eseguire query direttamente nel codice, semplificando la manipolazione e l'interrogazione di dati strutturati.

- **Interoperabilità**: *C#* offre una elevata interoperabilità con gli altri linguaggio supportati dall'ecosistema **.NET**, facilitando l'integrazione di componenti esistenti e la collaborazione con il codice scritto in diversi linguaggio, come ad esempio il linguaggio **C++**[23].

- **Retrocompatibilità**: Microsoft[21] fornisce un supporto solido ed aggiornamenti continui per *C#*, garantendo per la retro-compatibilità e consentendo agli sviluppatori di beneficiare delle nuove funzionalità senza dover riscrivere l'intero codice.

- **Community e Risorse**: *C#* gode di una community di sviluppatori molto vasta, e di un ricco ecosistema di risorse, tutorial e documentazione, semplificando il processo di apprendimento e risoluzione di problemi.

In conclusone, *C#* rappresenta una solida opzione per lo sviluppo di una Thing Directory, offrendo un ambiente di sviluppo robusto, moderno e ben supportato. La sua combinazione tra orientamento ad oggetti, sicurezza ed integrazione con **.NET**, lo rende un linguaggio più che adatto per la progettazione di sistemi complessi proprio come una Thing Directory.

4.2.2 JVM: Java

Java[16] è un linguaggio di programmazione ad alto livello, fortemente tipizzato ed orientato agli oggetti. Sviluppato da Sun Microsystems[24] nel 1995 ed ha guadagnato popolarità grazie alla portabilità e facilità di utilizzo.

Caratteristiche Principali

- **Orientamento agli Oggetti**: *Java*, così come *C#* è un linguaggio fortemente orientato agli oggetti, raggruppando il codice in concetti fondamentali come classi e oggetti. Anch'esso supporta i concetti di incapsulamento, ereditarietà e polimorfismo. Questo paradigma risulta particolarmente adatto per modellare entità complesse coinvolte nella Thing Directory.

- **Portabilità**: Una delle caratteristiche distintive di *Java* è la sua portabilità. Il codice sorgente *Java*, difatti, può essere scritto una sola volta ed eseguito su diversi sistemi operativi, grazie alla macchina virtuale **JVM**[18]. Questa permette di astrarre l'esecuzione del codice dal sistema operativo sottostante.
- **Gestione Automatica della Memoria**: *Java* utilizza un sistema di **garbage collection** per la gestione automatica della memoria, liberando gli oggetti non utilizzati, ottimizzando quindi l'utilizzo della memoria da parte dell'applicativo.
- **Multithreading**: *Java* supporta il mutithreading, consentendo l'esecuzione concorrente di più thread. Questo risulta utile per lo sviluppo di applicazioni in cui diverse attività devono poter essere eseguite contemporaneamente.
- **Sicurezza**: La **JVM** esegue il bytecode Java in un ambiente controllato, riducendo il rischio di esecuzione di codice potenzialmente dannoso.
- **Community Attiva**: *Java* ha una vasta ed attiva community di sviluppatori, comportando un supporto continuo, un enorme set di risorse ed una crescente quantità di librerie di terze parti.
- **Versioni e Aggiornamenti**: *Java* è in continua evoluzione con nuove evoluzioni che introducono miglioramenti e funzionalità.

In conclusone, *Java* proprio come *C#* rappresenta una solida opzione per lo sviluppo di una Thing Directory, offrendo un ambiente di sviluppo robusto, moderno e ben supportato. La sua combinazione tra orientamento ad oggetti, sicurezza e portabilità lo rende un linguaggio più che adatto per la progettazione di sistemi complessi come una Thing Directory.

4.2.3 JVM: Kotlin

Kotlin[20] è un linguaggio di programmazione moderno e conciso sviluppato da JetBrains nel 2011 come risposta alla necessità di un linguaggio più moderno e funzionale. È progettato per essere completamente interoperabile con *Java*, utilizzando la piattaforma **JVM**, permettendo agli sviluppatori di sfruttare le librerie e l'ecosistema *Java* esistenti, beneficiando delle caratteristiche innovative di *Kotlin*. Dalla sua introduzione ha guadagnato popolarità grazie alla sua sintassi chiara, alle funzionalità avanzate ed alla sua interoperabilità senza soluzione di continuità con *Java*.

Caratteristiche Principali

- **Sintassi Concisa**: Una delle caratteristiche distintive di *Kotlin* è la sua sintassi concisa. Il linguaggio è progettato per ridurre la verbosità del codice, consentendo la scrittura di codici più leggibili e mantenibili.

- **Null Safety**: *Kotlin* affronta il problema delle *null reference* con un approccio diretto, introducendo il concetto di *null safety*. Ciò significa che il tipo di una variabile deve essere dichiarato come **nullable** o **non-nullable**, riducendo gli errori legati a *null pointer*.

- **Paradigma Semi-Funzionale**: *Kotlin* si distingue per il suo paradigma di programmazione semi-funzionale, che offre agli sviluppatori un'alternativa più espressiva e flessibile per affrontare determinati problemi. questo paradigma consente di sfruttare concetti chiave della programmazione funzionale, integrandoli con gli aspetti più tradizionali della programmazione orientata agli oggetti.

- **Interoperabilità con Java**: *Kotlin* è completamente interoperabile con *Java*, il che significa che è possibile utilizzare le librerie Java già esistenti all'interno di *Kotlin*, chiamando codice *Java*, e viceversa. Questa caratteristica facilità l'adozione graduale di codice *Kotlin* in progetti *Java* già esistenti.

- **Extension Functions**: *Kotlin* supporta le **extension functions**, un concetto che consente agli sviluppatori di aggiungere nuove funzionalità a classi esistenti senza ricorrere all'ereditarietà. Questo promuove un design molto più flessibile ed estensibile.

- **Coroutines**: *Kotlin* introduce il concetto di **coroutines**, che mira a semplificare la gestione di operazioni asincrone senza dover ricorrere alla complessità dei *callback* o delle *promises*. Le **coroutines** semplificano la scrittura di codice asincrono più leggibile e mantenibile.

- **First-Class Citizens**: In *Kotlin* le funzioni sono del **first-class citizen**, ovvero possono essere assegnate a variabili, passate come argomenti ad altre funzioni e restituite come valori da altre funzioni. Questo permette una maggiore flessibilità nell'uso delle funzioni.

- **Smart Casts**: *Kotlin* introduce il concetto di **smart casts**, che permette il riconoscimento automatico dei tipi da parte del compilatore, nei costrutti condizionali *if* o *when*, evitando la necessità di effettuare casting espliciti.

- **Data Classes**: *Kotlin* semplifica la creazione di classi utilizzate principalmente per la memorizzazione di dati, creando le **data classes**. Queste classi generano automaticamente i metodi come *equals()*, *toString() hashCode()* e simili, semplificando la gestione dei dati immutabili.

In sintesi, *Kotlin* è un linguaggio di programmazione versatile, moderno e pragmatico che combina le caratteristiche migliori di altri linguaggi esistenti a soluzioni innovative.

L'interoperabilità con *Java* e le sue caratteristiche avanzate lo rendono una scelta interessante per gli sviluppatori che cercano un linguaggio più moderno e potente. Un particolare decisamente interessante è l'approccio semi-funzionale, che consente di sfruttare i paradigmi della programmazione funzionale in aggiunta a quelli della programmazione ad oggetti, offrendo un'esperienza di sviluppo più flessibile ed espressiva.

4.2.4 Benchmark

Nel contesto dello sviluppo di sistemi software, la valutazione ed il confronto delle performance di diversi linguaggi di programmazione è un aspetto cruciale per la scelta delle tecnologie più adatte ai problemi in questione. In questa sezione, saranno esposti e valutati i risultati di alcuni benchmark che confrontano le prestazioni dei tre linguaggi introdotti: *C#*, *Java* e *Kotlin*.
I benchmark sono stati progettati per misurare il tempo di esecuzione di determinati algoritmi, offrendo una valida indicazione delle prestazioni relative di ciascun linguaggio. I seguenti benchmark sono stati ottenuti da un'apposita **piattaforma online dedicata**[26], che mette a disposizione il codice ed i risultati ottenuti, per diversi linguaggi, maniera tale da consentire il confronto tra una moltitudine di linguaggi, oltre che offrire la possibilità di verificare il codice stesso.

C# - Java

Di seguito sono esposte le tabelle comparative dei risultati[27] ottenuti dai linguaggi Java e C#

lang	time	peak-mem	compiler/runtime
Java	672ms	412.4MB	openjdk 20
Java	799ms	666.8MB	graal/jvm 17.0.7
C#	1238ms	384.0MB	dotnet/aot 7.0.306

Tabella 4.1: Binary Trees - Input: 18

lang	time	peak-mem	compiler/runtime
C#	126ms	72.0MB	dotnet/aot 7.0.306
Java	165ms	108.4MB	openjdk 20
Java	187ms	164.5MB	graal/jvm 17.0.7

Tabella 4.2: Binary Trees - Input: 15

lang	code	time	peak-mem	compiler/runtime
C#	2.cs	730ms	94.3MB	dotnet 7.0.306
Java	1.java	2015ms	163.3MB	graal/jvm 17.0.7
Java	1.java	2133ms	200.5MB	openjdk 20
C#	1.cs	3945ms	38.7MB	dotnet 7.0.306
java	2.java	timeout	583.5MB	openjdk 20

Tabella 4.3: Mandelbrot - Input: 5000

lang	code	time	peak-mem	compiler/runtime
C#	2.cs	128ms	43.0MB	dotnet 7.0.306
Java	1.java	219ms	95.5MB	openjdk 20
C#	1.cs	228ms	35.5MB	dotnet 7.0.306
Java	1.java	242ms	101.2MB	graal/jvm 17.0.7
java	2.java	788ms	566.3MB	openjdk 20

Tabella 4.4: Mandelbrot - Input: 1000

lang	code	time	peak-mem	compiler/runtime
Java	1.java	521ms	286.6MB	openjdk 20
Java	1.java	573ms	370.7MB	graal/jvm 17.0.7
C#	2.cs	804ms	369.6MB	dotnet/aot 7.0.306
C#	1.cs	841ms	384.6MB	dotnet/aot 7.0.306
C#	2.cs	1026ms	413.9MB	dotnet 7.0.306
C#	1.cs	1047ms	441.4MB	dotnet 7.0.306

Tabella 4.5: MerkelTree - Input: 17

Dai risultati dei benchmark tra **Java** e **C#**, emerge un quadro interessante. I risultati esposti non mostrano una netta preferenza per uno dei due linguaggi per quanto riguarda l'aspetto prestazionale. La differenza tra i due linguaggi varia a seconda dei casi, con scenari in cui uno supera l'altro in termini di velocità di esecuzione o consumo di memoria ram. Si verificano casi in cui **Java** risulta essere più veloce e con un utilizzo inferiore di risorse, mentre in altre situazioni **C#** eccelle in termini di tempi di esecuzione. Alcuni casi evidenziano che, pur impiegando più tempo, un linguaggio potrebbe utilizzare significativamente meno risorse rispetto all'altro, o viceversa. Questa eterogeneità nei risultati sottolinea l'importanza di valutare attentamente le esigenze specifiche del progetto tenendo in considerazione vari aspetti, come prestazioni e consumo di risorse durante l'esecuzione.
La decisione tra **C#** e **Java** non può quindi basarsi unicamente su risultati di benchmark generici, ma richiede una valutazione mirata delle caratteristiche chiave stabilite

dal contesto applicativo.

Java - Kotlin

Il codice scritto nei linguaggi **Kotlin** e **Java** può operare sullo stesso runtime senza alcun problema, l'unica differenza che si presenta è il compilatore, il quale è individuale per i due linguaggi, ma che genera un binary eseguibile su JVM senza distinzioni.

Figura 4.1: JVM Interoperability

Di seguito sono esposte le tabelle comparative dei risultati[28] ottenuti dai linguaggi **Java** e **Kotlin**, confrontando quindi i due linguaggi che operano sulla **JVM**.

lang	time	peak-mem	compiler/runtime
Kotlin	639ms	412.9MB	kotlin/jvm 17.0.2
Java	672ms	412.4MB	openjdk 20
Java	799ms	666.8MB	graal/jvm 17.0.7
Kotlin	timeout	222.5MB	kotlin/native 1.8.21

Tabella 4.6: Binary Trees - Input: 18

lang	time	peak-mem	compiler/runtime
Kotlin	132ms	109.4MB	kotlin/jvm 17.0.2
Java	165ms	108.4MB	openjdk 20
Java	187ms	164.5MB	graal/jvm 17.0.7

Tabella 4.7: Binary Trees - Input: 15

lang	time	peak-mem	compiler/runtime
Kotlin	451ms	285.9MB	kotlin/jvm 17.0.2
Java	521ms	286.6MB	openjdk 20
Java	573ms	370.7MB	graal/jvm 17.0.7

Tabella 4.8: MerkelTree - Input: 17

lang	time	peak-mem	compiler/runtime
Kotlin	178ms	221.7MB	kotlin/jvm 17.0.2
Java	233ms	222.5MB	openjdk 20

Tabella 4.9: MerkelTree - Input: 15

Dalle tabelle mostrate emergono degli aspetti interessanti. Nel complesso, **Kotlin** ha dimostrato di essere più efficiente sia in termini di tempo di esecuzione che di utilizzo della memoria RAM. Nonostante in alcuni casi entrambi i linguaggi mostrano un consumo simile di memoria o differenze di tempo trascurabili, **Kotlin** ha costantemente dimostrato un vantaggio rispetto a **Java**, persino a parità di SDK Level.
Sulla base dei dati mostrati e della stretta interoperabilità con **Java**, **Kotlin** risulta essere una scelta preferibile rispetto a **Java** per l'implementazione della **Thing Directory**. I vantaggi in termini di efficienza e utilizzo della memoria, combinati con la possibilità di usufruire delle stesse librerie disponibili per **Java**, rendono **Kotlin** il candidato (per quanto riguarda linguaggi sui **JVM**) ideale per la realizzazione del progetto.

In conclusione, l'analisi dei benchmark ha rivelato una parità prestazionale tra i due linguaggi **C#** e **Kotlin**, non fornendo elementi decisivi per orientare una decisione tra i due. La decisione sulla selezione del linguaggio è difatti basata principalmente sulla quantità e disponibilità di librerie e risorse individuali dei linguaggi. È stato selezionato il linguaggio **Kotlin**, essendo risultato l'opzione più attraente, offrendo un vasto ecosistema di librerie, in particolare per quanto riguarda la gestione avanzata dei **Triple Store** e dei dati **RDF**, questo grazie all'interoperabilità già discussa con il linguaggio **Java**.
L'abbondanza di librerie a disposizione per **Kotlin** ha giocato un ruolo fondamentale nella scelta finale, offrendo una maggiore flessibilità e opzioni di sviluppo. Nella sezione successiva, esploreremo in dettaglio le specifiche librerie che sono state selezionate e

che hanno contribuito al successo generale del progetto, evidenziando come la ricchezza dell'ecosistema abbia influenzato la realizzazione del sistema.

4.3 Librerie

La realizzazione di un sistema di gestione di dati semanticamente arricchiti implica l'utilizzo di librerie specializzate per sfruttare appieno le potenzialità offerte dalle tecnologie coinvolte, come **RDF** [2.2.2], il **Triple Store**[2.2.2] ed il formato **JSON-LD**[2.1.2, i quali sono stati discussi in precedenza.
La progettazione e lo sviluppo di applicativi una **Thing Directory**, richiedono una solida infrastruttura software, la quale funge da fondamenta per la costruzione di Rest API semantiche avanzate. L'implementazione di tali API dipende fortemente dall'adozione di librerie specializzate al fine di semplificare ed ottimizzare il trattamento dei dati, in particolare quelli semanticamente arricchiti, facilitando aspetti cruciali come la manipolazione di Thing Description, la gestione di richieste HTTP concorrenti, sistemi di notifica e la memorizzazione dei dati.

Una **Thing Directory**, che funge da componente centrale di un ecosistema per il Web of Things, gestisce e distribuisce informazioni dettagliate sulle *things*, che possono rappresentare oggetti sia fisici che virtuali. La presenza di dati semanticamente strutturati, espressi attraverso standard come **RDF**[2.2.2] e **JSON-LD**[2.1.2], necessita l'impiego di librerie specializzate per consentire gestione e trasmissioni agevoli ed efficienti.

La selezione delle librerie è guidata da diversi criteri chiave, fondamentali per garantire il successo del progetto.:

- **Manipolazione e Gestione di JSON-LD ed RDF**: Devono essere selezionate librerie che consentano la creazione, l'aggiornamento e la rimozione di **Thing Description**, nonché la conversione tra i formati in causa (**JSON-LD** ed **RDF**).
- **Interoperabilità con il Web Semantico**: Un'integrazione agevole con le tecnologie del Semantic Web, compresa la gestione di Triple Store è necessaria.
- **Validazione Sintattica e Semantica**: Risulta **essenziale** la possibilità di effettuare una validazione accurata degli aspetti **semantici** e **sintattici** delle Thing Description, assicurando che siano rispettati gli standard semantici e sintattici definiti dal **W3C**[2].
- **Gestione delle Rest API**: Deve essere integrato il *routing* delle richieste, così come la gestione delle risposte e la creazione di un API che sia coerente, efficiente e ben strutturata.

In questa sezione, esploreremo le librerie selezionate, evidenziando come ciascuna di esse contribuisca a soddisfare i criteri delineati, fornendo quindi le fondamenta tecnologiche **essenziali** per la realizzazione della Thing Directory.

4.3.1 Server Framework: Ktor

Essendo il cuore pulsante di qualsiasi applicazione connessa, il server costituisce la componente radicale, orchestrando il flusso continuo di dati e funzionalità. La scelta di un framework che sia robusto e performante è fondamentale per la modellazione dell'architettura del server, e in questo ruolo fondamentale, **Ktor**[31] con l'engine **CIO**[32] emerge come una scelta che non solo serve a dar vita al server stesso, ma ne orchestra il funzionamento sfruttando appieno le potenzialità del linguaggio Kotlin.
Nello sviluppo lato server, la prima mossa è cruciale. Non si sta semplicemente scrivendo codice, ma si stanno definendo le fondamenta dell'intero progetto. Pertanto, la scelta della giusta struttura risulta una decisione importante per l'intero processo di sviluppo. Basti pensare che tutte le interazioni esterne con il progetto avverranno attraverso il server stesso, che dovrà quindi condurre orchestrazione delle diverse funzionalità: Endpoint delle **REST API**, logica di routing ed integrazione con componenti interne ed la gestione del servizio di **Server-Sent Events** (**SSE**)[17].
In questa orchestrazione, **Ktor**[31] non ha semplicemente un ruolo, ma è il pilastro, conducendo le coroutine asincrone attraverso il suo engine **CIO**[32].

In questa sezione, sarà approfondita la logica alla base della scelta di **Ktor** con l'engine **CIO** come spina dorsale del server. Sarà discusso come questo duo non solo dà vita alle REST API ed alla logica di routing, ma si armonizza senza sforzo con il panorama delle tecnologie web, garantendo efficienza, scalabilità ed estensibilità.

Ktor[31] è un framework moderno ed asincrono, sviluppato da **JetBrains**[25], su misura per la creazione di applicazioni web asincrone ed efficienti sviluppate con il linguaggio **Kotlin**. Abbracciando la sintassi concisa ed espressiva di **Kotlin**, **Ktor** offre una miscela unica di semplicità e potenza, rendendolo la scelta ideale per coloro che cercano un framework intuitivo per lo sviluppo di applicazioni web. Di seguito sono riportati i punti chiave del framework **Ktor**:

- **Natura Asincrona**: Al centro del design di **Ktor** c'è il supporto nativo per la programmazione asincrona, guidata dalle *coroutine* di **Kotlin**. Questo consente di scrivere codice *concorrente* e *non-blocking*, essenziale per la gestione efficiente di un gran numero di connessioni e richieste simultanee.

- **Modularità ed estensibilità**: **Ktor** adotta un'architettura modulare, consentendo una personalizzazione delle applicazioni aggiungendo o rimuovendo funzionalità in base alle necessità individuali dei progetti. Con questo aspetto minimalista, **Ktor** offre la flessibilità di incorporare esclusivamente i componenti necessari per un progetto specifico.

- **Sistema di Routing**: Il sistema di **routing** di **Ktor** è particolarmente espressivo e conciso, semplificando il processo di definizione degli endpoint e di gestione dei diversi metodi HTTP. Questo dona una maggiore leggibilità del codice, e contribuisce alla manutenibilità complessiva del progetto.

- **Negoziazione dei Contenuti**: **Ktor** semplifica la negoziazione dei contenuti, fornendo un supporto integrato per la gestione di diversi formati di dati, come ad esempio **JSON** o **XML**. Questo risulta particolarmente utile quando si creano delle API che devono comunicare con client diversi.

- **Autenticazione ed Autorizzazione**: **Ktor** semplifica l'implementazione dei meccanismi di autenticazione ed autorizzazione. Con i provider di autenticazione estensibili, si possono integrare facilmente molteplici metodi di autenticazione all'interno dei sistemi.

- **Supporto per le WebSocket**: **Ktor** include il supporto nativo per la comunicazione attraverso i **WebSocket**, facilitando la comunicazione bidirezionale ed in tempo reale tra client e server. Questo è fondamentale per le applicazioni che richiedono aggiornamenti o notifiche istantanei.

- **Community e Documentazione**: **Ktor** beneficia di un'enorme ed attiva community, e di una documentazione completa, fornendo ampie risorse agli sviluppatori per chiedere e ricevere assistenza, condividere conoscenze o esplorare funzionalità avanzate del framework.

Inoltre, una caratteristica interessante di **Ktor**, è la disponibilità di engines che offre, consentendo agli sviluppatori di selezionare l'engine che più rispecchia le caratteristiche ricercate per il proprio server. Di seguito sono riportate le alternative più popolari con le rispettive caratteristiche principali:

- **Netty**:
  - **Asincrono ed Alte Performance**: **Netty** è un framework popolare per la sua natura event-driven ed asincrona. Eccelle per gli scenari che richiedono performance elevate ed è un buon candidato per applicazioni con un grande numero di connessioni concorrenti.
  - **Scalabile**: **Netty**, con la sua architettura *non-blocking*, assicura la scalabilità, rendendolo ideale per progetti che richiedono la gestione di un numero significante di connessioni simultanee in maniera efficiente.
  - **Molte Features**: Offre una vasta gamma di feature, incluso il supporto per i WebSockets, rendendolo il candidato ideale per le applicazioni di comunicazione bilaterale.

- **Jetty**:
  - **Integrazione delle Servlet**: **Jetty** è un framework container per le *Servlet*, il che lo rende ampiamente adatto ed utilizzato per i progetti che richiedono la compatibilità con le strutture basate sulle servlet.
  - **Facilità di integrazione**: Offrendo il supporto alle Servlet, offre un ambiente familiare agli sviluppatori che hanno già esperienza con le *servlet Java*.
  - **Stabilità**: **Jetty** è noto anche grazie alla sua stabilità, ed è utilizzato ampiamente in ambienti di production. Questo lo rende una scelta affidabile per progetto che richiedono alta scalabilità.
- **TomCat**:
  - **Compatibilità con Servlet**: Simile a **Jetty**, **Tomcat** è un'altra opzione per gli sviluppatori che preferiscono la compatibilità con i container e le tecnologie basate su servlet.
  - **Standardizzazione**: **Tomcat** è un container di servlet ampiamente standardizzato. La scelta di questo engine garantisce la conformità alle specifiche e pratiche stabilite per le servlet.
  - **Community**: La community di **Tomcat** è ampia ed attiva, fornendo supporto ed aggiornamenti continui. Questo può risultare vantaggioso per i progetti che beneficiano di uno sviluppo guidato dalla community.
- **CIO (Coroutine-based I/O)**:
  - **Lightweight ed Asincrono**: L'engine **CIO** è basato sulle coroutine di Kotlin, ed è progettato per essere leggero ed operare in maniera asincrona. È particolarmente adatto per ambienti con risorse limitate e scenari in cui la reattività è fondamentale.
  - **Supporto alle Coroutine**: Sfruttando le coroutine di **Kotlin**, **CIO** semplifica la gestione del codice asincrono, promuovendo chiarezza, robustezza e manutenibilità del codice. Le coroutine offrono un approccio strutturato per gestire in modo efficiente le operazioni concorrenti.
  - **Adaptive Connection Pooling**: **CIO** include il pooling adattivo delle connessioni, adattandosi dinamicamente ai diversi carichi di lavoro. Questa particolare adattabilità ottimizza l'utilizzo delle risorse, rendendolo l'ideale per i modelli di comunicazione imprevedibili spesso riscontrato nelle applicazioni IoT.
- **ServletApplicationEngine**:

- **Compatibilità con le Servlet**: **ServletApplicationEngine** è progettato per integrarsi perfettamente con i servlet container esistenti, consentendo la distribuzione degli applicativi **Ktor** via servlet.
- **Flessibilità** È adatto per progetti in cui la compatibilità con altri servlet container, come Tomcat o Jetty, è un requisito fondamentale.
- **Migrazione ed Integrazione**: **ServletApplicationEngine** facilità la migrazione **graduale** delle applicazioni esistenti o l'integrazione con sistemi che si basano sulle servlet.

In sintesi, la flessibilità di **Ktor** nell'offrire più engines consente di prendere decisioni informate in base ai requisiti specifici del progetto. Che si dia priorità a prestazioni elevate, elaborazione asincrona e concorrente, alla compatibilità con le servlet o all'efficienza delle risorse, la selezione di engine di **Ktor** contente soluzioni su misura per diversi scenari applicativi.

L'engine che è stato selezionato per la realizzazione del progetto è **CIO**, suscitando particolare interesse grazie alle sue caratteristiche e alle possibilità di sperimentazione all'interno del codice tramite le funzionalità avanzate di Kotlin. Nella sezione che segue sarà discusso in maggior dettaglio l'engine **CIO**, illustrando ulteriori caratteristiche che hanno influito sulla decisione del suo utilizzo.

Engine: CIO

L'engine **CIO (Coroutine I/O)** all'interno del framework **Ktor** rappresenta l'evoluzione dello sviluppo di applicativi per il web, enfatizzando i paradigmi asincroni e sfruttando appieno la potenza delle coroutine di Kotlin. In questa sezione saranno approfondite le complessità di **Ktor CIO**, svelandone l'architettura, le capacità di integrazione, l'efficienza delle risorse e la moltitudine di funzionalità che lo rendono una scelta decisiva per gli applicativi web contemporanei.
Al centro dell'engine **CIO** si trova l'architettura asincrona, un cambiamento di paradigma fondamentale rispetto ai tradizionali modelli sincroni. Questa scelta progettuale garantisce che il motore possa gestire in modo efficiente numerose connessioni simultanee contemporaneamente. Rispetto alle controparti sincrone, la natura *non-blocking* di **CIO** migliora la reattività, rendendolo una risorsa inestimabile per gli applicativi che richiedono aggiornamenti in real-time, e gestione efficiente degli eventi asincroni. L'integrazione delle coroutine di Kotlin: La sinergia tra il motore CIO e le coroutine di Kotlin è una caratteristica distintiva che lo distingue nello sviluppo web asincrono. Le coroutine forniscono agli sviluppatori un tipo di approccio differente, strutturato e di alto livello per quanto riguarda la scrittura di codice asincrono. L'integrazione delle coroutine non solo semplifica la complessità associata ai sistemi basati su callback, ma migliora inoltre la leggibilità e la manutenibilità del codice, consentendo agli sviluppatori di esprimere in un

modo più intuitivo ed elegante le complessità della logica asincrona. Nel contesto delle applicazioni IoT, dove eventi e risposte sono intrinsecamente asincroni, il supporto delle coroutine all'interno dell'engine CIO rappresenta una funzionalità da non sottovalutare.
Un'altra caratteristica distintiva è l'impegno verso la leggerezza ed efficienza in termini di risorse. Questo attributo diventa di particolare importanza negli ambienti in cui le risorse sono limitate, come ad esempio nei dispositivi IoT, o negli scenari di edge computing. L'ingombro minimo di **CIO** garantisce un utilizzo ottimale delle risorse, rendendolo la scelta ideale per le applicazioni eseguite su dispositivi con potenza di elaborazione e memoria limitate, senza inficiare sulle prestazioni.
Inoltre, **CIO** introduce un approccio dinamico o adattivo di pooling delle connessioni, adattandosi in tempo reale ai differenti carichi di lavoro. Questo meccanismo di **Adaptive Connection Pooling** ottimizza l'utilizzo delle risorse, regolando in maniera dinamica il numero di connessioni in base alla domanda. Ed è proprio ciò a rendere efficiente l'utilizzo delle risorse di sistema, rendendo l'engine particolarmente adatto per applicazioni che presentano modelli di traffico fluttuanti o con improvvisi picchi di domanda.
La combinazione di architettura asincrona, coroutine e pooling adattivo, rende CIO una soluzione scalabile ad alte prestazioni. La capacità di gestire in modo efficiente un gran numero di connessioni simultanee lo rende un ottimo candidato per applicazioni con carichi di lavoro non prevedibili. CIO offre quindi prestazioni eccezionali, creando le basi per applicazioni che siano reattive e scalabili.
Un altro punto importante è l'estensibilità e l'integrazione di middleware. La sua architettura estensibile facilità notevolmente l'incorporazione di funzionalità e middleware personalizzati, dando la possibilità di adattare l'engine alle esigenze specifiche dei singoli progetti. Questa estensibilità offre strade di integrazione di protocolli specializzati, meccanismi di autenticazione o logiche di elaborazione personalizzate (sarà discusso in maggior dettaglio quest'ultimo punto all'interno del capitolo 5 in quanto sono state integrate funzionalità personalizzate per la gestione degli eventi). Questa flessibilità garantisce che l'engine **CIO** possa adattarsi ad una vasta gamma di requisiti applicativi.
In conclusione, l'engine **CIO** all'interno di Ktor emerge non solo come strumento di sviluppo web, ma come una vera e propria possibilità di trasformazione nel panorama delle applicazioni asincrone e ad alte prestazioni. La combinazione di architettura asincrona, integrazione di coroutine, efficienza delle risorse, pooling di connessioni adattivo ed estensibilità lo posiziona come una scelta strategica per progetti che variano tra IoT (come nel caso di questo progetto) e applicazioni/servizi web su larga scala. L'engine **CIO** rappresenta un approccio innovativo per soddisfare le sfide e le richieste dello sviluppo web contemporaneo, offrendo efficienza, scalabilità e reattività inaudite.

4.3.2 RDF & Triple Store

Apache Jena[33] è un'importante framework open-source progettato appositamente per gestire dati RDF e per semplificare lo sviluppo di applicazioni per il web semantico, offrendo inoltre la possibilità di integrare un Triple Store direttamente all'interno del progetto. Offre a disposizione un set ricco di funzionalità, che lo rendono una risorsa di inestimabile valore per i progetti che puntano a sfruttare i principi del **WoT**, migliorando l'interoperabilità dei dispositivi. Di seguito, le capacità chiave che hanno determinato l'uso di **Jena** all'interno del progetto:

- **Gestione Grafi RDF**: La caratteristica principale di **Apache Jena** risiede nella capacità di gestire i grafi **RDF**. Questa librerie gestisce ad-hoc la creazione, manipolazione e navigazione dei grafi **RDF**, consentendo la rappresentazione di relazioni e gerarchie complesse inerenti agli scenari WoT.

- **Supporto a SPARQL**: Oltre alla gestione del formato **RDF**, **Apache Jena** offre il supporto al linguaggio di query **SPARQL**[10], il quale permette di eseguire particolari query semantiche sui dati **RDF**. Ciò consente un'interrogazione dei dati **RDF** efficiente, facilitando l'estrazione di informazioni significative da fonti differenti e distribuite.

- **Embedded Triple Store**: Una delle caratteristiche più importanti di **Apache Jena** è la sua perfetta integrazione con i **Triple Store**. Difatti, offre la possibilità di integrare un **Triple Store** all'interno del progetto ospitante, funzionalità fondamentale per il progetto in questione, permettendo la gestione, il recupero, ma soprattutto, l'archiviazione di enormi volumi di dati **RDF**. **Apache Jena** consente infatti di ospitare i **TDB** (Triplestore Database) e **TDB2** (una versione aggiornata dei **TDB** che presenta novità e vantaggi a livello prestazionale) all'interno del server. Ciò significa che si ha a disposizione una soluzione di archiviazione locale, eliminando la necessità di componenti e connessioni esterne.

- **Estensibilità e Modularità**: L'architettura modulare di **Apache Jena** garantisce flessibilità ed estensibilità. Questo offre la possibilità di selezionare le componenti specifiche da integrare in base ai requisiti individuali del progetto, consentendo approcci su misura per la gestione dei dati **RDF** nelle applicazioni che ne fanno uso.

- **Community e Supporto**: **Apache Jena** ha una enorme community di sviluppatori a disposizione, oltre che offrire una documentazione esaustiva delle funzionalità che offre, rendendola una libreria semplice da utilizzare anche per chi ha i primi approcci con tale.

- **Tools di utilità**; Uno strumento interessante offerto da **Apache Jena** è **Fuseki**, una componente progettata per gestire efficientemente le query con dati RDF. Questo strumento fornisce un accesso diretto al motore di query di Jena, permettendo l'esecuzione di query SPARQL direttamente sul TDB, senza la necessità di interfacciarsi con ulteriori API. Idealmente questo strumento viene utilizzato per scopi di testing al fine di verificare che le API implementate eseguano correttamente le operazioni sullo store.

Di seguito viene riportata l'architettura[35] di **Apache Jena**:

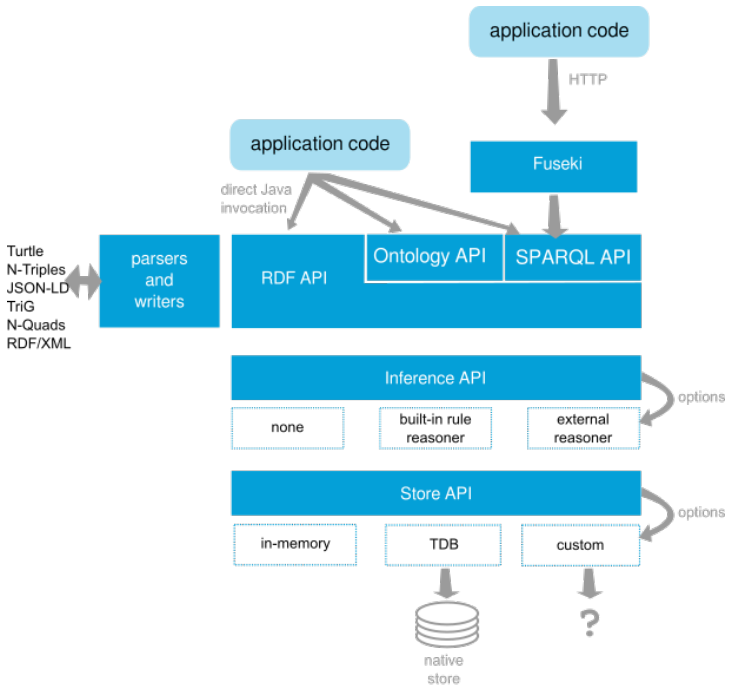

Figura 4.2: Jena Architecture[35]

Nell'ambito dello sviluppo WoT, la libreria **Apache Jena** rappresenta un potente strumento per la gestione di dati **RDF** e l'integrazione delle funzionalità di Triple Store. Il set di funzionalità offerte, le prestazioni robuste ed il supporto attivo della comunità lo rendono una risorsa indispensabile per applicazioni WoT che siano scalabili ed efficienti. È risultata fondamentale per la realizzazione del progetto grazie alla possibilità di integrare il **Triple Database** (**TDB** e **TDB2**)

4.3.3 JSON & JSON-LD

JSON

Come già accennato, una Thing Directory ha bisogno di lavorare costantemente con dati JSON, e la libreria selezionata per supportare questa necessità è **FasterXML Jackson**[37].
FasterXML Jackson è la libreria più famosa ed utilizzata per la gestione dei dati **JSON**, risultando lo standard per la serializzazione **JSON** per **Java** (e di conseguenza i linguaggi su JVM). Conosciuta per la sua velocità, flessibilità e facilità d'uso, Jackson è risultata perfetta per la realizzazione del progetto. Di seguito, le features più interessanti della libreria:

- **Data Binding**: Il cuore pulsante della libreria **Jackson** è l'**ObjectMapper**, un engine robusto per il mapping di dati da JSON a Object e viceversa. Questa funzionalità semplifica notevolmente i processi di conversione, permettendo di interagire con i dati JSON senza alcuna difficoltà.

- **Streaming API**: **Jackson** mette a disposizione i moduli **JsonParser** e **JsonGenerator**, delle potenti ed efficienti api di processing per stream JSON dalle dimensioni notevoli. Questa feature risulta particolarmente utile quando si deve interagire con dataset vasti, permettendo di effettuare *parsing* e *generation* incrementali di dati JSON.

- **Support alle Annotazioni**: Offre *annotazioni* come '**@JsonProperty**', permettendo agli sviluppatori di personalizzare il *mapping* tra **Objects** ed i campi di dati **JSON**. Questo controllo permette la gestione di strutture dati molto più complesse, assicurando un'integrazione fluida con codebase già esistenti.

- **Serialization e Deserialization custom**: Per scenari in cui il mapping di default non è sufficiente, **Jackson** mette a disposizione i moduli **JsonSerializer** e **JsonDeserializer**, permettendo la creazione di **serializer** e **deserializer** ad-hoc per le situazioni. Questa flessibilità assicura che anche le strutture dati più intricate e complesse possano essere supportate.

- **Performance ed Efficienza**: Le API di streaming richiedono un utilizzo minimo di memoria, rendendole ideali per scenari in cui le risorse sono da considerarsi un fattore critico.

- **Sviluppo Attivo e Mantenimento**: **Jackson** è attualmente in continuo sviluppo e manutenzione, con una community attiva che contribuisce alla sua evoluzione. Questo assicura una continua compatibilità con versioni e standard più recenti di Java e JVM.

JSON-LD

La libreria selezionata per la gestione delle funzionalità aggiuntive offerte dal formato **JSON-LD** è **Apicatalog Titanium**[38], una potente libreria sviluppata in Java, progettata appositamente per lavorare con i documenti in formato **JSON-LD** per permettere una interoperabilità semantica, con particolare attenzione per le prestazioni. Questa libreria permette il parsing, la manipolazione e serializzazione dei dati **JSON-LD**, coprendo tutte le funzionalità necessarie, ed offrendo una buona documentazione. L'aspetto più importante è che questa libreria rispetta le specifiche **W3C** per il formato **JSON-LD**, il che la rende perfetta per il progetto in questione. Alcune delle funzionalità di rilievo sono:

- **Context Handling**: **Titanium** supporta l'uso dei context di **JSON-LD**, permettendo di definire e gestire questa funzionalità chiave del formato, funzionalità **essenziale** per una Thing Directory.
- **Operazioni Compact, Expand, Frame e Flatten**: **Titanium JSON-LD** supporta tutte le operazioni di *compact*, *expand*, *frame* e *flatten* sui documenti **JSON-LD**, permettendo la conversione tra diverse rappresentazioni *linked-data*.
- **Conversione da e verso RDF**: Una feature particolarmente interessante di questa libreria, che l'ha resa il candidato ideale per il progetto, è il supporto alla conversione da JSON-LD a RDF e viceversa.

L'integrazione delle librerie per **Apicatalog Titanium** per la gestione dei dati JSON-LD e **Jackson** per la gestione dei dati JSON generici, accresce notevolmente le funzionalità e la robustezza della Thing Directory. Queste librerie giocano un ruolo fondamentale nella gestione dei metadati WoT, offrendo delle solide fondamenta per encoding e decoding efficienti e manipolazione delle Thing Description.

Capitolo 5

Implementazione & Testing

Dietro ogni progetto software vi sono una serie di decisioni cruciali, ciascuna delle quali contribuisce al successo complessivo e all'efficacia della soluzione. In questo capitolo saranno spiegate le scelte strategiche fatte durante la nascita ed evoluzione della Thing Directory WoTerFlow. Dalle strutture di archiviazione dei dati ai protocolli di interazione, fornendo informazioni sulla progettazione del progetto e sul suo allineamento con gli standard W3C.
Oltre discussioni teoriche, sarà mostrato codice, svelando frammenti e segmenti dell'implementazione. Questo capitolo sarà un'esplorazione pratica del codice reale che alimenta la Thing Directory, evidenziando non solo i comportamenti funzionali ma anche le soluzioni eleganti e creative, oltre che le ottimizzazioni che fanno risaltare tale implementazione.

5.1 API Routes

Un aspetto fondamentale della progettazione di un applicativo server, in questo caso una Thing Directory, è la definizione delle API Routes. Questi percorsi fungono da veri e propri sentieri digitali, guidando il flusso di informazioni tra la Thing Directory e gli utenti, consentendo una comunicazione e interazione senza soluzione di continuità.

In questa sezione esploreremo le API Routes accuratamente realizzate, e rispettando meticolosamente le linee guida del W3C, per facilitare lo scambio di dati all'interno dell'ecosistema di una Thing Directory. Questi **paths** non solo costituiscono una componente importante per una directory, ma svolgono anche un ruolo cruciale nel plasmare l'esperienza utente e nel garantire l'efficienza dell'intero sistema.
Di seguito saranno listati i percorsi, includendo informazioni riguardanti i metodi di comunicazione ed una descrizione della funzionalità che soddisfano:

API Endpoint	**Method**	**Descrizione**
/things	*HEAD*	Restituisce le informazioni header
/things/	*HEAD*	Restituisce solo le informazioni header
/things	*GET*	Restituisce il listing di tutte le TD registrate, in formato **JSON-LD**
/things/	*GET*	Restituisce il listing di tutte le TD registrate, in formato **JSON-LD**
/things{offset,limit,sort_by,sort_order}	*GET*	Restituisce il listing di tutte le TD registrate filtrando secondo i parametri richiesti, in formato **JSON-LD**
/things/{id}	*HEAD*	Verifica se una data TD esiste nel sistema
/things/{id}	*GET*	Restituisce la TD con l'*id* corrispondente al parametro inviato (se esistente), in formato JSON-LD
/things	*POST*	Crea una Anonymous TD con i parametri inoltrati nel body della richiesta (**DEVE** essere in formato **JSON-LD**). Restituisce l'id nel **Location Header** della risposta
/things/	*POST*	Crea una Anonymous TD con i parametri inoltrati nel body della richiesta (**DEVE** essere in formato **JSON-LD**). Restituisce l'id nel **Location Header** della risposta
/things/{id}	*PUT*	Crea (se non esistente) o Aggiorna (altrimenti) una TD con i parametri inoltrati nel body della richiesta (**DEVE** essere in formato **JSON-LD**). Restituisce l'id nel **Location Header** della risposta

/things/{id}	*PATCH*	Effettua l'aggiornamento parziale di una TD esistente con i parametri inoltrati nel body della richiesta (**DEVE** essere in formato **JSON-LD**). Restituisce l'id nel **Location Header** della risposta
/things/{id}	*DELETE*	Elimina una TD con l'id corrispondente al parametro della richiesta

Tabella 5.1: API Routes per la gestione delle Thing Description

API Endpoint	**Method**	**Descrizione**
/events	*GET*	Sottoscrive alle notifiche di **tutti** gli eventi (creazione, update ed eliminazione) supportati dal server. Gli eventi sono trasmessi via Server-Sent Events (**SSE**). Qualora una TD dovesse essere creata/modificata/eliminata sarà inviata una notifica tramite il canale SSE
/events/thing_created	*GET*	Sottoscrive alle notifiche degli eventi di creazione. Gli eventi sono trasmessi via Server-Sent Events (**SSE**). Qualora una TD dovesse essere creata sarà inviata una notifica tramite il canale SSE
/events/thing_updated	*GET*	Sottoscrive alle notifiche degli eventi di update. Gli eventi sono trasmessi via Server-Sent Events (SSE). Qualora una TD dovesse essere modificata sarà inviata una notifica tramite il canale SSE
/events/thing_deleted	*GET*	Sottoscrive alle notifiche degli eventi di eliminazione. Gli eventi sono trasmessi via Server-Sent Events (SSE). Qualora una TD dovesse essere eliminata sarà inviata una notifica tramite il canale SSE

Tabella 5.2: API Routes per la sottoscrizione agli eventi

API Endpoint	Method	Descrizione
/search/sparql	*HEAD*	Risponde alla richiesta **head** sulla route per SPARQL Semantic Search
/search/sparql{query}	*GET*	Esegue la ricerca di Thing Description tramite una query SPARQL (rispettando gli standard SPARQL 1.1). I risultati possono essere restituiti nei seguenti formati, a seconda dell'header della richiesta (JSON come default). **ASK** e **SELECT**: *JSON* (application/sparql-results+json), *XML* (application/sparql-results+xml), *CSV* (text/csv), *TSV* (text/tab-separated-values). **CONSTRUCT** o **DESCRIBE**: *TURTLE* (text/turtle)
/search/sparql{query}	*POST*	Esegue la ricerca di Thing Description tramite una query SPARQL (rispettando gli standard SPARQL 1.1). I risultati possono essere restituiti nei seguenti formati, a seconda dell'header della richiesta (JSON come default). **ASK** e **SELECT**: *JSON* (application/sparql-results+json), *XML* (application/sparql-results+xml), *CSV* (text/csv), *TSV* (text/tab-separated-values). **CONSTRUCT** o **DESCRIBE**: *TURTLE* (text/turtle)
/search/jsonpath{query}	*GET*	Esegue la ricerca di Thing Description tramite una query JSONPath (rispettando gli standard JSONPath). I risultati sono restituiti nel formato *JSON*
/search/xpath{query}	*GET*	Esegue la ricerca di Thing Description tramite una query XPath (rispettando gli standard XPath 3.1). I risultati sono restituiti nel formato *JSON*

Tabella 5.3: API Routes per utilizzare le query di ricerca avanzate semantiche/sintattiche

5.2 Archiviazione Dati

Nel perseguimento di un archivio dati che sia efficiente e stabile per l'implementazione della Thing Directory, è stata ideata una struttura di archiviazione ibrida. Questo approccio combina i punti di forza di un TDB2 persistente (Triple Database v2) con una HashMap in memoria. L'obiettivo è sfruttare le capacità di archiviazione persistente di TDB2 per garantire la durabilità, sfruttando contemporaneamente la ricerca rapida O(1) fornita da una HashMap durante il runtime. Questa struttura ibrida garantisce prestazioni e reattività ottimali, mantenendo un discreto trade-off in termini di uso della memoria, trovando un equilibrio tra l'esigenza di durabilità e la richiesta di un rapido recupero dei dati.
Discutiamo ora i dettagli di ciascuna componente della struttura ibrida dell'archivio dati.

5.2.1 Triple-Store: Apache Jena

Grazie all'utilizzo della libreria **Apache Jena** è possibile usufruire del suo modulo TDB2, il quale copre la funzione di storage *locale* persistente per le Thing Description. La sua struttura supporta le triple RDF, offrendo la possibilità di effettuare querying e retrieval efficienti tramite query SPARQL. Il TDB2 agisce come una 'source of truth', riflettendo lo stato più aggiornato dei metadati. Quando vengono inoltrati nuovi dati al sistema, essi vengono prima di tutto validati, ed una volta verificatane la correttezza, vengono inseriti all'interno del TDB2 per garantirne persistenza. Anche nel caso della modifica viene effettuato un controllo sulla validità dei dati prima di inserirli nel TDB2.
Il vantaggio principale è la persistenza, rendendolo uno storage ideale anche per dati che richiedono di integrità e longevità all'interno del sistema. Questo assicura che lo stato più recente delle Thing Description sia preservato anche nell'eventualità di shutdown o restart del sistema.

5.2.2 In-Memory: HashMap

Per accelerare le operazioni di ricerca durante il runtime, viene utilizzata una HashMap per conservare in memoria le descrizioni delle Things. Questa struttura dati garantisce una complessità lineare O(1), fornendo un rapido accesso ai metadati per quanto riguarda le operazioni che richiedono esclusivamente la lettura dei dati (anche chiamate operazioni di **lookup**), evitando inoltre ulteriori accessi non necessari allo store persistente TDB2. L'HashMap viene caricata dinamicamente all'avvio del sistema, creando un indice in memoria di tutte le Thing Description esistenti. Quando nuovi dati vengono aggiunti al TDB2, questi vengono successivamente caricati nella HashMap, garantendo che la rappresentazione in memoria rimanga sincronizzata con l'archiviazione persistente. Questa sincronizzazione garantisce che l'HashMap rifletta costantemente le ultime aggiunte o modifiche ai metadati.

5.2.3 I Vantaggi

L'implementazione ibrida offre un equilibrio tra durabilità ed efficienza di runtime. La persistenza di TDB2 garantisce la robustezza dei dati, mentre la HashMap soddisfa l'esigenza di un rapido lookup dei dati durante il runtime. I vantaggi principali dell'approccio ibrido sono:

- **Durabilità e Coerenza**: Il TDB2 garantisce che i metadati siano mantenuti in modo coerente e durevole nel tempo.
- **Efficienza in Runtime**: L'HashMap in-memory fornisce accesso rapido alle Thing Description durante il runtime, contribuendo a risposte con latenza il più bassa possibile, e prestazioni ottimali del sistema.

5.2.4 Il Compromesso

Sebbene l'approccio sacrifichi parte della RAM per l'HashMap in-memory, questo compromesso è stato considerato accettabile visti i vantaggi sostanziali in termini di prestazioni e reattività. Il sacrificio della RAM è una decisione consapevole per ottimizzare l'esperienza complessiva e garantire un rapido accesso ai metadati delle Thing Description.

Nel complesso, l'implementazione ibrida di TDB2 e HashMap nella Thing Directory raggiunge un equilibrio ottimale tra durabilità ed efficienza di runtime. TDB2 garantisce un'archiviazione persistente, mentre HashMap facilita le operazioni di lookup degli elementi. Questa combinazione strategica è una scelta deliberata per migliorare le prestazioni complessive, anche con un compromesso selettivo nell'utilizzo della memoria RAM, rendendola una soluzione solida ed efficiente per la gestione di metadati per Thing Descriptions.

5.2.5 Il Codice

Creazione ed Update degli archivi

Di seguito è mostrato il codice per collegare la Thing Directory agli archivi:

- **TDB2**:
  - Viene controllata l'esistenza della directory locale contenente il TDB2 al path "data/tdb-data", e nel caso non dovesse essere presente, viene creata.
  - Vengono caricati tutti i dati presenti nella suddetta directory.
- **HashMap**:

```
Utils.createDirectoryIfNotExists("data/tdb-data")
val rdf_db: Dataset = TDB2Factory.connectDataset("data/tdb-data")

val thingsMap: MutableMap<String, ObjectNode> = ConcurrentHashMap()

...

val ts = ThingDescriptionService(rdf_db, thingsMap)
ts.refreshJsonDb()
```

Listing 2: Creazione/Lettura archivi

- Viene creata una HashMap utilizzando il costrutto ConcurrentHashMap, così da consentire accesso in lettura concorrente (per velocizzare richieste di lookup concorrenti).
- Vengono aggiunti tutti i valori presenti all'interno del TDB2, così da avere a disposizione la copia necessaria.

```
fun refreshJsonDb() {
    rdfDataset.begin(TxnType.READ)

    try {
        val ttlList = Utils.loadRDFDatasetIntoModelList(rdfDataset)
        val things = ttlList.map {
            converter.toJsonLd11(converter.fromRdf(it)) }

        // Clear the things map and populate it back with the updated dataset
        thingsMap.clear()

        // Update the in-memory cache with the refreshed JSON representation
        things.forEach { thing ->
            thing["id"]?.asText()?.let { id ->
                thingsMap[
                    Utils.strconcat(DirectoryConfig.GRAPH_PREFIX, id)
                ] = thing
            }
        }

        Utils.printWatermark()
    } catch (e: Exception) {
        throw ThingException("Error refreshing the JsonDb: ${e.message}")
    } finally {
        rdfDataset.end()
    }
}
```

Listing 3: Caricamento di tutte le TD all'interno della HashMap

Questo codice è necessario per effettuare il caricamento di tutti i dati presenti all'interno del TDB2 nella HashMap. Viene eseguito esclusivamente all'avvio della Thing Directory, avendo un carico computazionale proporzionato al numero di TD presenti nel TDB2.

- Vengono letti i dati del TDB2.
- Le TD vengono convertite in **JSON-LD** tramite il costrutto *map* ed una classe **Converter** appositamente implementata per effettuare le conversioni tra i diversi formati a disposizione.
- Per ogni elemento nella lista, si selezionano l'*id* ed il contenuto completo, che verranno quindi inseriti nella HashMap.

Di seguito, una variante del codice precedente, che performa il refresh di un singolo elemento della HashMap. Utilizzato quando viene inserito, modificato o eliminato un elemento, effettuando l'aggiornamento del singolo, evitando il reload dell'intero TDB2.

```
private fun refreshJsonDbItem(graphId: String) {
    try {
        val ttlModel = Utils.loadRDFModelById(rdfDataset, graphId)

        if (!ttlModel.isEmpty){
            val objNode = converter.fromRdf(ttlModel)
            val thing = converter.toJsonLd11(Utils.toJson(objNode.toString()))

            //  Update the in-memory cache with the refreshed JSON representation
            thingsMap[graphId] = thing
        } else {
            thingsMap.remove(graphId)
        }
    } catch (e: Exception) {
        throw ThingException(
            "Error refreshing the JsonDb item with id: $graphId: ${e.message}")
    }
}
```

Listing 4: Update di un singolo elemento all'interno della HashMap

- Si legge il nuovo contenuto della Thing Description
- Si effettua la conversione a JSON-LD
- Il valore nella HashMap è aggiornato attraverso l'*id* della TD, il quale è univoco.
- Nel caso non vi fosse alcuna modifica, bensì una rimozione, esso viene rimosso dalla HashMap.

Anonymous Thing Description Insert

In questa sezione viene mostrato il funzionamento dell'insert di una Thing Description. Di seguito viene riportato quindi il codice di una Thing Description *Anonima*, ovvero senza il campo '*@id*' necessario per identificarla all'interno di un Triple Store. Questo tipo di Thing Description serve per inserire un nuovo elemento all'interno del Triple Store, senza specificarle l'*id*, in maniera tale da concedere la totale libertà di selezionarne uno da parte della Thing Directory:

```
{
    "@context": [
        "https://www.w3.org/2019/wot/td/v1"
    ],
    "security": [
        "nosec_sc"
    ],
    "securityDefinitions": {
        "nosec_sc": {
            "scheme": "nosec"
        }
    },
    "title": "example thing"
}
```

Listing 5: Esempio di JSON-LD contenente una Thing Description anonima (senza id)

Questa Thing Description contiene le informazioni minime necessarie per permettere la registrazione, come *'@context'* e *'title'*, oltre alle informazioni relative allo schema di sicurezza (in questo caso 'nosec', ovvero assente).

Il processo di registrazione di una Thing Description include l'inserimento di informazioni, come ad esempio i timestamp di creazione e modifica, il tipo, la versione e altri eventualmente specificati dal sistema che effettua la registrazione.
La stessa TD, dopo essere registrata tramite WoTerFlow, apparirà come segue:

```json
{
    "id": "urn:uuid:84528a33-f706-48f1-8037-cde0d17f4dbb",
    "@type": "Thing",
    "security": [
        "nosec_sc"
    ],
    "securityDefinitions": {
        "nosec_sc": {
            "scheme": "wotsec:NoSecurityScheme"
        }
    },
    "registration": {
        "created": "2023-12-18T17:11:05.072527600Z",
        "modified": "2023-12-18T17:11:05.072527600Z"
    },
    "title": "example thing",
    "@context": "https://www.w3.org/2022/wot/td/v1.1",
    "@version": "1.1"
}
```

Listing 6: Esempio di JSON-LD contenente una Thing Description dopo la registrazione

Di seguito viene invece riportato il codice *Kotlin* che implementa una funzione con transazione per eseguire l'Insert di una Thing Description

```kotlin
private inline fun Dataset.createWithTransaction(action: () ->
    Pair<String, ObjectNode>) : Pair<String, ObjectNode> {

    this.begin(TxnType.WRITE)
    return try {
        val result = action()
        this.commit()
        result
    } catch (e: Exception) {
        this.abort()
        throw e
    } finally {
        this.end()
    }
}
```

Listing 7: Codice creazione Anonymous Thing via transaction

Questo codice presenta l'esempio di una **extension function**, inserendo una funzione nuova all'interno della classe *Dataset*. Inoltre, questo codice permette l'inserimento dei dati attraverso una *transaction*, ovvero, nel caso parte del codice dovesse fallire, fallirebbe l'intero inserimento della Thing Description all'interno dello store persistente.
La struttura di questa funzione, permette di eseguire all'interno di esse un'ulteriore funzione, passata tramite l'argomento *action*.
Il risultato restituito dalla funzione estensione è una coppia contenente l'*id* generato per la TD e la TD stessa con l'aggiunta dei dati relativi alla registrazione, come l'id ed i timestamp di creazione, in maniera tale da restituire la TD registrata al client che ha richiesto l'inserimento.

```
fun insertAnonymousThing(td: ObjectNode): Pair<String, ObjectNode> {
        var mapId: String = ""
        synchronized(this) {
            return runCatching {
                rdfDataset.createWithTransaction {
                    val pair = generateUniqueID()
                    val id = pair.first
                    val graphId = pair.second

                    td.put("@id", id)

                    //  Checking the jsonld version and upgrading if needed
                    val tdVersion11p = Utils.isJsonLd11OrGreater(td)
                    val tdV11 = if (!tdVersion11p) converter.toJsonLd11(td)
                        else td

                    //  JsonLd decoration with missing fields
                    decorateThingDescription(tdV11)

                    //  Model Validation
                    val jsonRdfModel = converter.toRdf(tdV11.toString())
                    validateThingDescription(jsonRdfModel)

                    //  Query preparation for RDF data storing
                    val rdfTriplesString = converter.toString(jsonRdfModel)
                    val query = """
                        INSERT DATA {
                            GRAPH <$graphId> {
                                $rdfTriplesString
                            }
                        }
                    """.trimIndent()

                    // Execute query
                    SparqlService.update(query, rdfDataset)

                    mapId = graphId
                    Pair(id, tdV11)
                }
            }.onSuccess {
                refreshJsonDbItemIfNotEmpty(mapId)
            }.getOrElse { e ->
                throw handleException(e, "Insert Anonymous Thing")
            }
        }
    }
```

Listing 8: Codice creazione Anonymous Thing

Questa funzione completa l'implementazione del codice necessario per l'inserimento della Thing Description, facendo uso della *extension function* precedentemente presentata [7] per sfruttare il meccanismo di transazione.
All'interno di questo codice, si evidenzia l'impiego della keyword '*synchronized*', che crea un blocco sincronizzato, garantendo che solo un thread possa eseguire un certo blocco di codice in un dato momento. Questo meccanismo è fondamentale per prevenire accessi concorrenti a risorse condivise, eliminando così potenziali **race conditions** ed assicurando la **thread safety**. In questo contesto è necessario per controllare l'accesso alle risorse di archiviazione, mettendo a disposizione solo la versione più recente (e coerente) di ogni Thing Description.
Il corpo della funzione esegue le seguenti operazioni:

- Genera un **uuid** e lo assegna alla TD.
- Verifica la versione della TD, per determinare come interagire con essa.
- applica una "decorazione" alla TD, aggiungendo dati di registrazione specifici.
- Esegue la **validazione** della TD secondo gli schemi di validazione ufficiali [39] [40].
- Prepara la query per l'inserimento nel Triple Store e la esegue.
- Restituisce la coppia (UUID, TD)
- **In caso di Successo**: aggiorna il valore della TD all'interno della HashMap.
- **In caso di Fallimento**: gestisce l'exception terminando la transazione e restituendo i valori relativi al fallimento.

L'ultima componente utilizzata per l'inserimento di una Thing Description è la funzione che provvede all'esecuzione di una query **SPARQL**. Questa funzione è necessaria per interfacciarsi al TDB, eseguendo la funzione passata come argomento.

```
fun update(query: String, dataset: Dataset) {
    try {
        val updateQuery = UpdateFactory.create(query)
        val updateProcessor = UpdateExecutionFactory.create(updateQuery, dataset)

        updateProcessor.execute()
    } catch (e: Exception) {
        throw Exception("${e.message}")
    }
}
```

Listing 9: Codice per l'esecuzione di una query di update del TDB2

Grazie al codice illustrato è quindi possibile effettuare l'insert di una nuova Thing Description all'interno del TDB. Per quanto riguarda le altre operazioni, come update, patch e rimozione, il codice subisce sottili modifiche a seconda dell'operazione necessaria, in particolare, la modifica principale è la query SPARQL che sarà poi inoltrata alla funzione apposita[9].
Di seguito sono presentate le query per le diverse operazioni:

```
val query = """
    INSERT DATA {
        GRAPH <$graphId> {
            $rdfTriplesString
        }
    }
""".trimIndent()
```

Listing 10: Query per l'insert di una Anonymous Thing Description

```
val query = """
    DELETE WHERE {
        GRAPH <$graphId> {
            ?s ?p ?o
        }
    };
    INSERT DATA {
        GRAPH <$graphId> {
            $rdfTriplesString
        }
    }
""".trimIndent()
```

Listing 11: Query per l'update e patch di una Thing Description

```
val query = "DELETE WHERE { GRAPH <$graphId> { ?s ?p ?o } }"
```

Listing 12: Query per la rimozione una Thing Description

Per quanto invece riguarda il **retrieve** delle Thing Description, come già annunciato, non è necessario interfacciarsi con il TDB utilizzando query, bensì è sufficiente interrogare la HashMap in base al tipo di ricerca vogliamo effettuare.
Ad esempio, per ottenere tutte le Thing Description registrate, si utilizza la seguente funzione, che restituisce semplicemente tutti i valori caricati in memoria:

```
fun retrieveAllThings(): List<ObjectNode> {
    try {
        return thingsMap.values.toList()
    } catch (e: Exception){
        throw ThingException("Retrieve All: ${e.message}")
    }
}
```

Listing 13: Codice per la restituzione di tutte le Thing Description presenti

Mentre, per effettuare l'operazione di lookup per ID, viene utilizzata la seguente funzione, che sfrutta la complessità computazionale O(1) offerta dalla HashMap:

```kotlin
fun retrieveThingById(id: String): ObjectNode? {
    try {
        val graphId = Utils.strconcat(DirectoryConfig.GRAPH_PREFIX, id)
        return thingsMap[graphId]
    } catch (e: Exception){
        throw ThingException("Retrieve Get: ${e.message}")
    }
}
```

Listing 14: Codice per la ricerca tramite ID

5.3 Server-Sent Events (SSE)

Nel campo dello sviluppo web, la comunicazione in tempo reale tra server e client può rivelarsi essenziale per creare applicazioni dinamiche e reattive. Un potente meccanismo che facilita questo flusso di informazioni è il protocollo **Server-Sent Events** (**SSE**). **SSE** è una soluzione leggera ed efficiente per inviare dati dai server ai client attraverso una singola connessione HTTP.
Server-Sent Events è un protocollo semplice e standardizzato costruito sulla base del tradizionale protocollo HTTP. A differenza di altre tecnologie di comunicazione in tempo reale, SSE è caratterizzato dalla sua semplicità, facilità d'uso ed integrazione con tecnologie web esistenti. L'idea alla base di SSE è quella di consentire ai server di inviare aggiornamenti in tempo reale ai client attraverso una singola connessione HTTP di lunga durata.

Le caratteristiche principali di SSE:

- **Comunicazione unidirezionale**: **SSE** segue un modello di comunicazione unidirezionale, in cui un client instaura una connessione con il server, il quale successivamente si occuperà di inoltrare gli aggiornamenti al client. Questo lo rende particolarmente adatto a scenari in cui il server deve informare i client su eventi, come aggiornamenti di notizie, variazioni di azioni o altri dati in tempo reale.

- **Formato testuale**: I messaggi **SSE** sono inviati in un formato testuale, che li rende leggibili dall'uomo e semplici da debuggare. Il payload è tipicamente codificato in UTF-8, consentendo la trasmissione di vari tipi di dati, tra cui **JSON**.

- **Riconnessione automatica**: Le connessioni **SSE** tentano automaticamente di riconnettersi in caso di disconnessione, garantendo la resistenza alle fluttuazioni della rete. Questa funzione garantisce agli utenti un'esperienza di comunicazione in tempo reale più solida.

- **Formato del flusso di eventi**: I messaggi **SSE** sono organizzati in un formato di flusso di eventi. Ogni evento è costituito da una serie di campi, tra cui il nome dell'evento, il payload e gli identificatori opzionali. Il formato stream consente di raggruppare logicamente i messaggi correlati e aiuta i clienti a distinguere i diversi tipi di eventi.

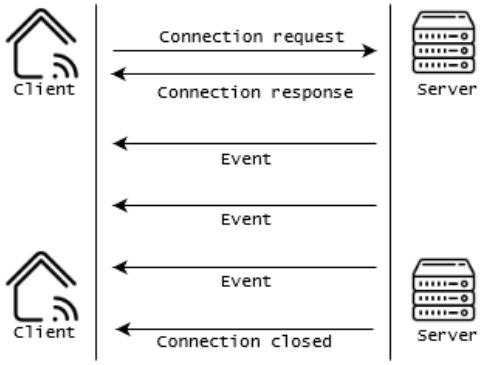

Figura 5.1: Comunicazione via SSE

Nel contesto del Web of Things, **SSE** svolge un ruolo cruciale nel facilitare la comunicazione in tempo reale tra dispositivi e applicazioni abilitati al WoT, fornendo un meccanismo leggero ed efficiente per la trasmissione di aggiornamenti e notifiche dai server ai client. In particolare possiamo individuare alcuni punti di interesse:

- **Aggiornamento delle Thing Description in tempo reale**: Con l'integrazione di **SSE**, una Thing Directory può notificare in tempo reale gli aggiornamenti dai client quando vi sono dei cambiamenti nelle Thing Description, assicurando che tutti i client siano sempre al corrente delle versioni più recenti di ogni TD.

- **Event Notification**: I dispositivi WoT possono generare eventi in base a varie condizioni o trigger. SSE può quindi essere utilizzato per notificare ai client questi eventi in tempo reale. Ciò è particolarmente utile per gli scenari in cui è fondamentale una risposta immediata agli eventi, come le letture dei sensori o i cambiamenti di stato dei dispositivi.

- **Scalabilità**: Gli ambienti WoT possono coinvolgere un gran numero di dispositivi e client. L'implementazione di SSE deve quindi essere il più scalabile possibile per poter gestire il volume potenzialmente elevato di eventi in tempo reale, mantenendo le prestazioni e la reattività.

L'integrazione dei Server-Sent Events migliora l'ecosistema WoT, favorendo un'interazione più dinamica e reattiva tra i dispositivi WoT e le applicazioni che sfruttano le loro capacità.

5.3.1 Kotlin: Flow e MutableSharedFlow

Nel panorama dinamico dello sviluppo lato server, adottare i giusti modelli di programmazione concorrente ed asincrona è cruciale. Kotlin introduce un potente costrutto all'interno della sua libreria standard, il **Flow**. Questo costrutto si integra perfettamente con le coroutines e le funzionalità asincrone di framework come Ktor [4.3.1], fornendo una solida base per l'implementazione di funzionalità di comunicazione in tempo reale proprio come **SSE**.

Flow è un costrutto per l'elaborazione reattiva dei flussi, che consente l'elaborazione asincrona e sequenziale dei dati. Fornisce un approccio dichiarativo al lavoro con i flussi di dati asincroni, il che lo rende una scelta ideale per gli scenari in cui gli eventi devono essere gestiti in modo asincrono ed efficiente.

Le caratteristiche principali di **Flow** sono:

- **Asincrono e Non-Blocking**: Flow supporta la programmazione asincrona senza il blocco dei thread, consentendo una gestione efficiente di attività concorrenti. Questa natura non-blocking è particolarmente vantaggiosa nelle applicazioni server-side, dove la reattività e l'utilizzo delle risorse sono fondamentali.

- **Dichiarativo e Composable**: La natura dichiarativa dei **Flow** permette ai programmatori di esprimere operazioni asincrone complesse in modo chiaro e conciso. Questo promuove la componibilità, consentendo la creazione di componenti di codice modulari e riutilizzabili per gestire vari aspetti dell'elaborazione asincrona dei dati.

- **Cancellazione e Gestione delle Eccezioni**: **Flow** fornisce meccanismi per gestire le cancellazioni e le eccezioni in modo semplice. Questo assicura che le risorse siano rilasciate in modo appropriato e che le condizione di errore siano gestite nel modo corretto.

Nel contesto di una Thing Directory, l'integrazione di Flow, in particolare di MutableSharedFlow, diventa particolarmente vantaggiosa per l'implementazione dei Server-Sent Events. MutableSharedFlow agisce come una fonte condivisa di eventi asincroni, rendendolo ideale per la trasmissione di aggiornamenti a più subscribers, come i client connessi in tempo reale.

I vantaggi di Flow per l'implementazione di SSE in una Thing Directory sono:

- **Concorrenza e Scalabilità**: La natura non-blocking dei Flow consente di gestire in modo efficiente le connessioni SSE simultanee Questo è particolarmente importante nelle Thing Directory, dove più client possono sottoscrivere aggiornamenti simultaneamente.

- **Integrazione perfetta con Ktor**: Ktor, essendo un framework moderno ed asincrono per Kotlin, si integra perfettamente con i Flow. Questa sinergia semplifica l'implementazione degli endpoint SSE e garantisce una comunicazione fluida tra server e client.
- **Codice Espressivo per la Gestione degli SSE**: La natura dichiarativa ed espressiva di Flow consente di creare codice pulito e conciso per la gestione degli eventi SSE. Si possono modellare facilmente i flussi di eventi e e trasformazioni, ottenendo un codice particolarmente mantenibile e comprensibile
- **Supporto integrato per la cancellazione**: Il supporto integrato di **Flow** per la cancellazione si allinea bene con i requisiti di SSE, dove i client possono connettersi e disconnettersi dinamicamente. Questo assicura che le risorse siano gestite in modo appropriato, evitando potenziali perdite di risorse.

5.3.2 Il Codice

Il framework **Ktor** non supporta nativamente il protocollo SSE, ed è stato quindi necessario implementarlo. Utilizzando le funzionalità offerte da Kotlin, come i **Flow** e le **extension function**, è stato possibile implementare questa funzionalità in modo semplice ed intuitivo, favorendo un'ottima leggibilità.

I canali di comunicazione da supportare all'interno di una Thing Directory, come visibile nella tabella [5.2] sono i seguenti:

- ***thing_created***: Quando viene inserita (o creata) una nuova Thing Description all'interno del sistema.
- ***thing_updated***: Quando viene modificata una Thing Description già presente all'interno del sistema.
- ***thing_deleted***: Quando viene rimossa una Thing Description dal sistema.
- ***all***: Non viene specificato alcuna canale e vengono quindi notificati tutti i tipi di evento.

È importante notare che uno stesso client può voler richiedere la subscription per più tipi di eventi, ma non tutti, ad esempio necessita sapere solo quando le Thing Description vengono create ed eliminate.

Come prima cosa bisogna definire la struttura che forma un messaggio SSE. Sappiamo che un messaggio SSE è composto da:

- ***data***: il payload di dati che si vuole trasmettere.

- ***event***: il tipo di evento che si vuole notificare (facoltativo).
- ***id***: codice identificativo dell'evento (facoltativo).

Proseguiamo quindi a creare una *data class* per racchiudere questi dati:

```
data class SseEvent(
    val data: String,
    val event: String? = null,
    val id: String? = null
)
```

Listing 15: Codice per la definizione della data class SseEvent

Definiamo ora la **extension function** per integrare la funzionalità di inoltro di messaggi SSE, estendendo la classe ApplicationCall di Ktor.
Come argomenti della funzione vi sono:

- **eventList**: Una lista di eventi SSE, necessaria per inoltrare eventi precedenti nel caso in cui il client volesse recuperare tutti gli eventi dopo un dato ID.
- **eventsFlow**: Una lista di Flows, utilizzata per inoltrare gli eventi nel flow destinazione.

```kotlin
suspend fun ApplicationCall.respondSse(
                    eventsList: List<SseEvent>,
                    vararg eventsFlows: Pair<EventType, Flow<SseEvent>>) {

    with(response) {
        header(HttpHeaders.CacheControl, "no-cache")
        header(HttpHeaders.Connection, "keep-alive")
        status(HttpStatusCode.OK)
    }

    try {
        respondBytesWriter(contentType = ContentType.Text.EventStream) {
            eventsList.forEach { event ->
                writeEvent(event)
            }

            eventsFlows.map { it.second }
                .merge()
                .collect { event ->
                    writeEvent(event)
                }
        }
    } catch (e: ChannelWriteException) {
        throw InternalError(e.message)
    }
}
```

Listing 16: Extension Function che implementa il respondSse nella classe ApplicationCall

Nel codice possiamo vedere come prima di tutto si inoltrano tutti gli eventi presenti in eventList, fornendo quindi tutti gli eventi successivi al desiderato. Successivamente qualora un nuovo evento dovesse entrare all'interno del Flow, sarà inoltrato al client tramite la funzione ***writeEvent***.

Anche la funzione ***writeEvent*** è una **extension function**, che si occupa di preparare il payload dell'evento, per poi inoltrarlo sulla connessione stabilita con il client.

```kotlin
private suspend fun ByteWriteChannel.writeEvent(event: SseEvent) {
    val eventString = buildString {
        event.event?.let { append("event: $it\n") }
        event.id?.let { append("id: $it\n") }
        event.data.lines().forEach { append("data: $it\n") }
        append("\n")
    }
    writeStringUtf8(eventString)
    flush()
}
```

Listing 17: Extension Function writeEvent

In fine vi è una classe apposita **EventController** che si occupa della gestione degli eventi.
Viene definita passando come argomento i Flow per i tre tipi di eventi, in maniera tale da avere accesso diretto a tali, ed inizializzando due variabile:

- **pastEvents**: La lista contenente la cronologia di tutti gli eventi avvenuti.
- **idCounter**: Un contatore per tener traccia dell'id assegnato all'ultimo evento fino al momento corrente.

```kotlin
class EventController(val thingCreatedSseFlow: MutableSharedFlow<SseEvent>,
                      val thingUpdatedSseFlow: MutableSharedFlow<SseEvent>,
                      val thingDeletedSseFlow: MutableSharedFlow<SseEvent>
) {
    private val pastEvents = CopyOnWriteArrayList<SseEvent>()
    private val idCounter = AtomicLong(0)
    ...
}
```

Listing 18: Definizione classe EventController

All'interno di questa classe possiamo trovare le seguenti funzioni.

La funzione addEvent compone un SseEvent [15] e lo inserisce all'interno della lista ***pastEvents***, per poi restituirlo alla funzione chiamante.

```kotlin
private fun addEvent(eventType: EventType, eventData: String): SseEvent {
    val event = SseEvent(
        data = eventData,
        event = eventType.toString().toLowerCasePreservingASCIIRules(),
        id = idCounter.getAndIncrement().toString()
    )

    pastEvents.add(event)
    return event
}
```

Listing 19: Funzione addEvent

La funzione **getPastEvents** permette di ottenere tutti gli eventi presenti nella lista degli eventi già avvenuti, dopo un dato ID facoltativo. Per ottimizzare le performance di tale funzione, viene prima di tutto controllata la presenza di eventi nella lista **pastEvents**, in modo tale da fermare immediatamente l'esecuzione della funzione nel caso non vi siano elementi, per poi proseguire con la selezione degli elementi successivi all'id indicato. In caso non vi siano elementi all'interno della lista cronologica, viene semplicemente restituita una lista vuota.

```kotlin
fun getPastEvents(lastReceivedId: String?, vararg eventTypes: EventType):
    List<SseEvent> {

    // Early return if pastEvents is empty or the lastReceivedId is null
    if (pastEvents.isEmpty() || lastReceivedId == null)
        return emptyList()

    // Filter past events by EventType and ID
    return lastReceivedId.toLongOrNull()?.let { lastId ->
        pastEvents.filter { event ->
            (event.id?.toLong() ?: 0) > lastId &&
                EventType.fromString(event.event.toString()) in eventTypes
        }
    } ?: emptyList()
}
```

Listing 20: Funzione getPastEvents

La funzione redirectEventFlow permette di reindirizzare l'invio di un dato evento al Flow che raccoglie esclusivamente gli eventi del tipo corrispondente. Viene utilizzato il costrutto **when** per trarre vantaggio dalla sintassi di Kotlin.

```
private suspend fun redirectEventFlow(eventType: EventType, event: SseEvent) {
    when (eventType) {
        EventType.THING_CREATED -> {
            thingCreatedSseFlow.emit(event)
        }
        EventType.THING_UPDATED -> {
            thingUpdatedSseFlow.emit(event)
        }
        EventType.THING_DELETED -> {
            thingDeletedSseFlow.emit(event)
        }
    }
}
```

Listing 21: Funzione redirectEventFlow

Per finire, vi è la funzione **notify**, la quale ha l'utilità di permettere la notifica di uno specifico evento da classi esterne all'**EventController**, utilizzando la funzione **redirectEventFlow** per il corretto reindirizzamento dell'evento.

```
suspend fun notify(eventType: EventType, eventData: String) {
    val event = addEvent(eventType, eventData)

    redirectEventFlow(eventType, event)
}
```

Listing 22: Funzione notify

Di seguito è riportato un esempio di connessione in listening per tutti i tipi di evento, tramite il software Postman[41].

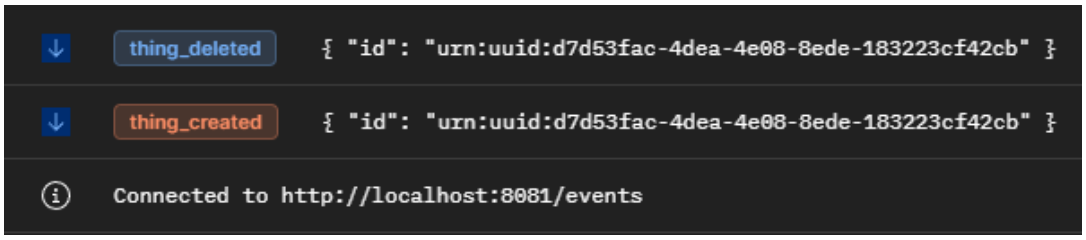

Figura 5.2: SSE listening tramite Postman[41]

In questo esempio è stata prima di tutto stabilita una connessione tramite l'url *http://localhost:8081/events*, richiedendo quindi di essere notificati riguardo tutti i tipi di evento. Dopodiché è stata creata un Thing Description, la cui creazione è stata notificata, per poi successivamente eliminarla, ricevendo la notifica di eliminazione sempre sullo stesso canale di comunicazione SSE.

5.4 Advanced Search

Questa sezione approfondisce le complessità della funzionalità di ricerca avanzata, un'aggiunta rivoluzionaria che consente agli utenti di navigare nel regno del WoT con una maggiore precisione. Il cuore di questo miglioramento è costituito dalla possibilità di effettuare ricerche avanzate tramite tre linguaggi di query: **JSONPath**, **XPath** e **SPARQL**. Ognuno di questi linguaggio porta con se i propri punti di forza, consentendo agli utenti di creare query complesse che estraggono, filtrano e recuperano informazioni rilevanti dalle vastità di una Thing Description Directory.

5.4.1 Syntactic Search: JSONPath

Iniziando da JSONPath, un linguaggio di query leggero ed intuitivo, progettato per attraversare e manipolare le strutture JSON senza sforzo. Approfondendo la sua sintassi e le sue capacità, sveliamo il potenziale che possiede per semplificare le ricerche, offrendo agli utenti un'esperienza senza soluzione di continuità nell'estrazione di informazioni preziose dalla directory.
Per implementare le funzionalità di ricerca tramite query JSONPath, è stato fatto uso della libreria apposita JayWay JSONPath[42], una libreria opensource e molto promettete per quanto riguarda le performance.
Inoltre, un'ottimizzazione non indifferente è l'utilizzo della HashMap [5.2.2] per quanto riguarda la ricerca tramite query JSONPath, approfittando della velocità di tale struttura dati e del caricamento in memoria già pronto per effettuare le ricerche al massimo delle prestazioni possibile.
Di seguito è riportato il codice responsabile per l'esecuzione delle query JSONPath:

```kotlin
fun executeQuery(query: JsonPath, map: Map<String, ObjectNode>):
    List<ObjectNode> {

    val jsonPathConfig = Configuration.defaultConfiguration()

    return map.values.filter { td ->
        val matchingNodes = JsonPath.using(jsonPathConfig)
            .parse(td.toString())
            .read(query) as List<ObjectNode>

        matchingNodes.isNotEmpty()
    }
}
```

Listing 23: Funzione executeQuery per JSONPath

Questa funzione filtra la mappa di ObjectNodes basandosi sul risultato della query JsonPath ottenuto dalla libreria JayWay JsonPath, verificando se vi sono match. La funzione restituisce quindi una lista di ObjectNode contente i risultati della query. Il filtraggio è ottenuto effettuando il parsing di ogni ObjectNode come string JSON, e applicando la query JsonPath, per poi controllare se tutti i risultati ottenuti non siano nulli.

5.4.2 Syntactic Search: XPath

Il secondo metodo di ricerca Sintattica è tramite l'utilizzo del linguaggio di query XPath, uno standard collaudato per la navigazione e interrogazione di documenti XML. Il suo utilizzo permette di estendere le abilità di attraversamento della Thing Directory utilizzando le descrizioni basate su XML. L'utilizzo di questo linguaggio permette di aggiungere complessità alle espressioni di interrogazione, attraversando la struttura gerarchica dei documenti XML, permettendo di raffinare le ricerche ed isolare le informazioni più pertinenti.
Anche le query XPath sono effettuate utilizzando la HashMap, eseguendo una conversione da JSON a XML, per poi utilizzare una libreria ad-hoc per l'esecuzione di query XPath sui dati appena convertiti. La libreria in questione è Saxon HE[43], una libreria opensource e disponibile sia per linguaggi su JVM che .NET, la libreria più longeva ed utilizzata per le query XPath.
Il codice per implementare questa funzionalità di ricerca è più complesso rispetto a JSONPath, richiedendo molteplici componenti, per rendere il codice più modulare ed interpretabile.

Come prima cosa, sono dichiarati il Processor ed il Compiler, ovvero gli organi che si occuperanno di eseguire le query. Nel blocco **init** viene impostata la versione del linguaggio da utilizzare, come richiesto dalle specifiche W3C, pari a **3.1**.

```
class XPathService {
    companion object{
        private val processor = Processor(false)
        private val compiler = processor.newXPathCompiler()

        init {
            compiler.languageVersion = "3.1"
        }

        ...
    }
}
```

Listing 24: Funzione per la composizione di un documento XML

È quindi necessario comporre un documento XML partendo da una stringa.

```
private fun buildXmlDocument(xmlString: String): XdmNode {
    val documentBuilder = processor.newDocumentBuilder()
    val xmlSource = StreamSource(StringReader(xmlString))

    return documentBuilder.build(xmlSource)
}
```

Listing 25: Funzione per la composizione di un documento XML

La seguente funzione esegue la query richiesta su un singolo documento XML, per verificare se conforme o meno alla query.

```
private fun evaluateXPath(query: String, tdDocument: XdmNode): XdmValue {
    val xpathSelector = compiler.compile(query).load()
    xpathSelector.contextItem = tdDocument

    return xpathSelector.evaluate()
}
```

Listing 26: Funzione per l'esecuzione della query su un singolo documento XML

Abbiamo quindi l'estrazione di una Thing Description dalla map contenente tutte le TD rilevanti, in base alla proprietà ID.

```
private fun extractMatchingTD(results: XdmValue, map: Map<String, ObjectNode>):
    ObjectNode? {

    if (results is XdmNode) {
        val resultJson = convertXdmToJson(results)
        return if (resultJson is ObjectNode) {
            resultJson["id"]?.textValue().let {
                map[Utils.strconcat(DirectoryConfig.GRAPH_PREFIX, it!!)] ?:
                    throw NoSuchElementException("No Such Element: id -> $it")
            }
        } else {
            null
        }
    }
    return null
}
```

Listing 27: Funzione per l'estrazione delle TD corrispondenti

La seguente funzione è necessaria per effettuare la conversione da XML a JSON

```
private fun convertXdmToJson(xdmNode: XdmNode): JsonNode {
    val objectMapper = XmlMapper()
    val jsonText = xdmNode.toString()
    return objectMapper.readTree(jsonText)
}
```

Listing 28: Funzione per effettuare conversione da XML a JSON

In fine, la funzione che esegue la query sull'insieme di Thing Description presenti nella memoria. Questa orchestra l'utilizzo di tutte le componenti precedentemente illustrate.

```
fun executeQuery(query: String, map: Map<String, ObjectNode>): List<ObjectNode> {
    return map.values
        .mapNotNull { td ->
            val jsonNode = ObjectMapper().valueToTree<JsonNode>(td)
            val xmlString = XmlMapper().writeValueAsString(jsonNode)
                .replace("@", "")
            val tdDocument = buildXmlDocument(xmlString)
            val results = evaluateXPath(query, tdDocument)
            extractMatchingTD(results, map)
        }
}
```

Listing 29: Funzione executeQuery per XPath

5.4.3 Semantic Search: SPARQL

Per quanto riguarda la ricerca semantica, è utilizzato il linguaggio di query SPARQL. Questo linguaggio opera sui documenti RDF, permettendo agli utenti di effettuare query semantiche esprimendo relazioni tra entità. Nel contesto delle Thing Description, SPARQL può essere utilizzato per formulare query che vanno ben oltre il syntactic matching, permettendo la scoperta di dispositivi basandosi su relazioni semantiche, ontologiche e contestuali.
Nel caso della ricerca semantica tramite SPARQL, a differenza della ricerca tramite JSONPath e XPath, le query vengono eseguite direttamente sul TDB attraverso le API offerte da Jena. È stato comunque necessario implementare del codice per effettuare le dovute conversioni ed operazioni per rendere i risultati fruibili da servizi web. Di seguito il codice implementato.

La funzione estensione writeTransaction, simile ad altre già viste (7), permette di eseguire il codice all'interno di una transaction.

```
private fun Dataset.writeTransaction(action: Dataset.(Dataset) ->
    ByteArrayOutputStream): ByteArrayOutputStream {

    begin(TxnType.WRITE)
    try {
        return action(this, this)
    } finally {
        commit()
        end()
    }
}
```

Listing 30: Funzione estensione per consentire il read/write tramite transaction per le query SPARQL

La funzione **executeAskQuery** permette di restituire i risultati ottenuti dalla query nel formato selezionato dal cliente che richiede l'esecuzione della query di tipo *ASK*.

```
private fun executeAskQuery(queryExecution: QueryExecution, format: ResultsFormat,
    outputStream: ByteArrayOutputStream) {

    when (format) {
        ResultsFormat.FMT_RS_JSON -> ResultSetFormatter.outputAsJSON(outputStream,
            queryExecution.execAsk())
        ResultsFormat.FMT_RS_XML -> ResultSetFormatter.outputAsXML(outputStream,
            queryExecution.execAsk())
        ResultsFormat.FMT_RS_CSV -> ResultSetFormatter.outputAsCSV(outputStream,
            queryExecution.execAsk())
        ResultsFormat.FMT_RS_TSV -> ResultSetFormatter.outputAsTSV(outputStream,
            queryExecution.execAsk())
        else ->
            throw UnsupportedSparqlQueryException(
                                        "Unsupported format for ASK query")
    }
}
```

Listing 31: Funzione executeAskQuery per eseguire le query **SPARQL** di tipo **ASK**, restituendo il formato desiderato tra **JSON**, **XML**, **CSV**, **TSV**

Per le query di tipo *SELECT*, esiste la funzione **executeSelectQuery** il cui codice è molto simile alla seguente, ma con una differenza esclusivamente nel tipo di *exec* che

effettua, per cui sarà usato **execSelect()** anziché **execAsk()**.

Bisogna invece gestire in maniera diversa le query di tipo **CONSTRUCT** o **DESCRIBE**, le quali restituiranno un output esclusivamente nel formato **TURTLE**.

```
1  private fun executeConstructOrDescribeQuery(queryExecution: QueryExecution,
2      outputStream: ByteArrayOutputStream) {
3
4      val resultModel = if (queryExecution.query.isConstructType) {
5          queryExecution.execConstruct()
6      } else {
7          queryExecution.execDescribe()
8      }
9      resultModel.write(outputStream, Lang.TURTLE.name)
10 }
```

Listing 32: Funzione executeConstructOrDescribeQuery per eseguire le query SPARQL di tipo CONSTRUCT o DESCRIBE, restituendo il formato TURTLE

Per concludere vi è la funzione executeQuery, che permette di filtrare il tipo di operazione da eseguire.

```
fun executeQuery(query: String, format: ResultsFormat, dataset: Dataset):
    ByteArrayOutputStream {

    return dataset.writeTransaction {
        val outputStream = ByteArrayOutputStream()
        val queryExecution = QueryExecutionFactory.create(
            QueryFactory.create(query, Syntax.syntaxSPARQL_11), dataset)

        try {
            when {
                queryExecution.query.isAskType ->
                    executeAskQuery(queryExecution, format, outputStream)
                queryExecution.query.isSelectType ->
                    executeSelectQuery(queryExecution, format, outputStream)
                queryExecution.query.isConstructType ||
                    queryExecution.query.isDescribeType ->
                    executeConstructOrDescribeQuery(queryExecution, outputStream)
                else -> throw UnsupportedSparqlQueryException(
                                                    "Unsupported query type")
            }
        }finally {
            queryExecution.close()
        }

        outputStream
    }
}
```

Listing 33: Funzione executeQuery per SPARQL

In conclusione, le funzionalità di ricerca avanzata fornite da JSONPath, XPath e SPARQL, offrono agli utenti una gamma di strumenti per modellare e personalizzare le proprie query variando dalle semplici ricerche sintattiche, alla più avanzata esplorazione semantica, la quale copre molte più sfumature. Questi linguaggi di query consentono agli utenti di scoprire in modo efficiente i dispositivi e più in generale le Thing Description in linea con i propri requisiti specifici, contribuendo a un'esperienza WoT e IoT più fluida e incentrata sulle richieste individuali.

5.5 Testing

Nel regno dinamico delle Thing Description Directory di W3C, dove le intricate interconnessioni tra dispositivi definiscono il panorama del Web of Things, l'affidabilità ed interoperabilità di queste entità digitali sono fondamentali. Man mano che si sviluppano

le complessità di questo ecosistema, aumenta anche la necessità di un quadro di test solido e completo. Il test diventa il fulcro, garantendo che le specifiche delineate dal W3C non siano solo costrutti teorici ma componenti pratici, affidabili e perfettamente integrati nell'ecosistema.

Questo capitolo approfondisce l'ambito critico dei test all'interno del framework Thing Description del W3C, dove la convergenza di standard, protocolli e applicazioni del mondo reale richiede un esame meticoloso. Il viaggio attraverso i test non è semplicemente una fase del ciclo di vita dello sviluppo, bensì è un processo continuo che coinvolge ogni fase, dalla concezione iniziale del codice all'integrazione completa di diverse entità.

All'interno di questo capitolo esploriamo i principi fondamentali del test, sottolineando l'adozione di un approccio test-first che va di pari passo con l'etica dello sviluppo *agile*. Mentre si procede con la creazione ed il perfezionamento delle implementazioni della TDD, l'atteggiamento test-first diventa una guida, garantendo che ogni riga di codice si allinei con gli standard generali, rafforzando così le basi del software nel complesso. Questa metodologia *agile* non solo accelera la tempistica di sviluppo, ma permette di evitare l'accumulo di errori, garantendo che ogni aspetto delle specifiche W3C sia sottoposto a un controllo rigoroso in ogni fase.

5.5.1 Approccio: Test-First

L'adozione di un approccio test-first è fondamentale nello sviluppo e nell'implementazione delle specifiche W3C Thing Description Directory. Questa metodologia enfatizza la scrittura di test prima del codice vero e proprio, garantendo che l'implementazione soddisfi i criteri predefiniti di funzionalità e compatibilità. Aderendo a questa filosofia, gli sviluppatori possono convalidare sistematicamente il proprio codice rispetto a una serie di aspettative predefinite, promuovendo una cultura di affidabilità e prevedibilità nell'ecosistema.

L'implementazione dell'approccio test-first non solo aiuta a rilevare potenziali problemi nelle prime fasi del ciclo di sviluppo, ma fornisce anche una solida base per le fasi successive. Agisce come misura proattiva per individuare discrepanze e deviazioni allo standard, migliorando così la qualità complessiva delle implementazioni della Thing Description Directory W3C.
L'approccio test-first facilita lo sviluppo iterativo, in cui vengono apportati miglioramenti e modifiche garantendo al tempo stesso che la funzionalità esistente rimanga intatta, promuovendo un'implementazione stabile e resiliente.

5.5.2 Testing Suite

I test di integrazione svolgono un ruolo fondamentale nel convalidare la perfetta collaborazione di vari componenti all'interno del framework TDD. Viene utilizzata una serie di test per valutare l'operabilità e l'interazione tra diverse entità, garantendo che siano conformi agli standard e alle linee guida specificati.
Per condurre i test sul software, è stata impigata una libreria opensource sviluppata con il linguaggio Golang. Questa suite, aderisce alle rigide specifiche W3C, dimostrando di essere uno strumento completo ed affidabile per l'esecuzione di integration testing. In particolare, la suite è progettata per coprire l'intera gamma di test richiesti per garantire la conformità agli standard W3C.
Ciò che rende questa suite particolare è la sua origine e validazione all'interno del Working Group di WoT. Dopo aver contattato il gruppo di lavoro responsabile, è stato confermato che la stessa suite viene utilizzata in modo ufficioso all'interno del gruppo stesso, non esistendo **al momento** una suite ufficiale messa a disposizione dal W3C. Tale riconoscimento aggiunge un ulteriore livello di affidabilità e prestigio alla suite, sottolineando la sua idoneità per test di questo livello.
Oltre a soddisfare le esigenze standard di testing, presenti sulla repository ufficiale W3C [45], la suite si distingue grazie all'inclusione di altri test addizionali mirati a verificare funzionalità più specifiche e particolari delle componenti del software. Questi test supplementari sono progettati per esplorare approfonditamente le capacità del sistema, garantendo che anche aspetti meno estesi siano valutati.
In sintesi, l'adozione di questa suite opensorce ha rappresentato un passo cruciale nel garantire qualità e robustezza del codice, e grazie all'approvazione informale da parte del working group WoT, ne è confermata la sua validità e rilevanza nel contesto degli standard e pratiche del settore.
La suite in questione è ***Farshidtz wot-discovery-testing***[44], presente su GitHub e sviluppata nel linguaggio Go. Questa suite permette di effettuare test tu tutte le funzionalità essenziali e non, incluso il sistema di notifiche tramite protocollo SSE e ricerche avanzate con JSONPath, XPath e SPARQL, motivi per cui ha suscitato particolare interesse. Oltre a ricoprire tutti i test richiesti, include la scrittura di un documento riportante tutti gli esiti, positivi e negativi, indicando i risultati dei singoli test-case.

5.5.3 Test della Ricerca Avanzata

In questa sezione saranno illustrati i test sulle funzionalità di ricerca avanzata offerti dalla suite di testing. Per ogni linguaggio di query viene effettuato il test tramite una query, e viene verificato che i risultati ottenuti siano conformi all'esito positivo atteso.

Syntactic Search: JSONPath

Il test la ricerca sintattica via query JSONPath consiste nella seguente serie di operazioni:

- Vengono inserite delle Thing Description contenenti un campo particolare ***tag***, il quale sarà il soggetto utilizzato per la query.
- Viene effettuata una richiesta al server, richiedendo l'esecuzione della query di ricerca **JSONPath**, per filtrare ed ottenere esclusivamente le TD con il campo ***tag*** utilizzato in precedenza.
- Sono controllati gli elementi ottenuti e viene registrato l'esito dei test.

Di seguito è riportata la query utilizzata dalla suite per effettuare il test sulla ricerca sintattica con JSONPath, dove il valore ***tag_value*** viene sostituito dal tag generato in precedenza.

```
1 $[?(@.tag=='tag_value')]
```

Listing 34: JSONPath query utilizzata per il test.

Syntactic Search: XPath

Per la ricerca sintattica via query XPath, le operazioni sono simili alla ricerca tramite JSONPath:

- Sono inserire delle Thing Description contenenti un campo particolare ***tag***, il quale sarà il soggetto utilizzato per la query.
- Viene effettuata una richiesta al server, richiedendo l'esecuzione della query di ricerca **XPath**, per filtrare ed ottenere esclusivamente le TD con il campo ***tag*** utilizzato in precedenza.
- Sono controllati gli elementi ottenuti e viene registrato l'esito dei test

La query utilizzata per effettuare i test sulla ricerca sintattica con XPath è la stessa utilizzata per JSONPath. La differenza tra le due è l'endpoint su cui viene inoltrata la richiesta. Di seguito, la query utilizzata, dove il valore ***tag_value*** viene sostituito dal tag generato in precedenza.

```
$[?(@.tag=='tag_value')]
```

Listing 35: XPath query utilizzata per il test.

Semantic Search: SPARQL

Nonostante la suite sia completa, i test per la ricerca avanzata tramite SPARQL utilizzano query non ideali per le scelte progettuali di WoTerFlow. Questo poiché le query all'interno della suite vengono effettuate su TDB la cui architettura non supporta i **Named Graphs**. Siccome per WoTerFlow è stato deciso di supportare tale funzionalità, sono state necessarie modifiche alla suite, **esclusivamente per questo dettaglio**.
Di seguito sono riportate le query prima e dopo la modifica per il supporto ai Named Graphs. Queste query effettuano la selezione delle Thing Description, limitando il numero di entry da considerare a 5.

```
select * { ?s ?p ?o } limit 5
```

Listing 36: SPARQL query prima della modifica

```
SELECT ?s ?p ?o ?g WHERE { GRAPH ?g { ?s ?p ?o } } LIMIT 5
```

Listing 37: SPARQL query dopo della modifica

Oltre alla query già illustrata, sono stati eseguiti test con ulteriori query tramite il software **Postman**[41]. Questi test hanno permesso di verificare in profondità il funzionamento della funzionalità di ricerca semantica avanzata. Sono quindi riportate le query supplementari:

La seguente query effettua il retrieve delle prime 5 risorse create in un determinato orario.

```
SELECT ?s ?p ?o ?g
WHERE {
  GRAPH ?g {
    ?s <http://purl.org/dc/terms/created> \n
        "YYYY-MM-DDTHH:MM:SS.SSSZ"^^<http://www.w3.org/2001/XMLSchema%23dateTime> .
    ?s ?p ?o
  }
}
LIMIT 5

```

Listing 38: SPARQL query per ricerca in base al momento di creazione

La seguente query effettua il conteggio delle risorse il cui campo ***type*** contiene il valore "***Thing***".

```
SELECT (COUNT(?s) AS ?count)
WHERE {
  GRAPH ?g {
    ?s a <https://www.w3.org/2019/wot/td%23Thing> .
  }
}
```

Listing 39: SPARQL query per effettuare il conteggio di risorse in base valore del campo ***type***

e di seguito è riportato un esempio di risultato ottenuto dalla query precedente

```
{
    "head": {
        "vars": [
            "count"
        ]
    },
    "results": {
        "bindings": [
            {
                "count": {
                    "type": "literal",
                    "datatype": "http://www.w3.org/2001/XMLSchema#integer",
                    "value": "207"
                }
            }
        ]
    }
}
```

Listing 40: Risultato ottenuto dalla query 39

Vediamo ora una query per effettuare una ricerca più approfondita, chiedendo la ricerca di Thing Description che abbiano più valori specifici, come il ***tab*** ed il ***title***

```
SELECT ?s ?p ?o ?g
WHERE {
  GRAPH ?g {
    ?s <https://www.w3.org/2019/wot/td%23tag> \n
        "ab801f07-a28f-4be6-9452-5880c8d7c070" .
    ?s ?p ?o .
    ?s <https://www.w3.org/2019/wot/td%23title> "test thing"@en .
  }
}
```

Listing 41: SPARQL query per effettuare ricerca di Thing Description con molteplici valori specifici

In fine viene mostrata una query simile alla precedente, che però utilizza la keyword **OPTIONAL**, che permette di includere argomenti non vincolanti per la ricerca:

```
SELECT ?s ?p ?o ?g
WHERE {
  GRAPH ?g {
    ?s <https://www.w3.org/2019/wot/td%23tag> \n
        "ab801f07-a28f-4be6-9452-5880c8d7c070" .
    ?s ?p ?o .

    OPTIONAL {
       ?s <https://www.w3.org/2019/wot/td%23title> "test thing"@en .
        ?s ?p ?o
    }
  }
}
```

Listing 42: SPARQL query con keyword OPTIONAL

Conclusione

I risultati ottenuti dalla suite di testing non sono stati solo promettenti, ma anche indicativi delle capacità del sistema di interpretare e rispondere in modo efficace a query complesse, dimostrando le capacità di ricerca migliorate della Thing Description Directory WoTerFlow.

Oltre alla suite di test, sono stati introdotti ulteriori casi di test, per sfidare ulteriormente i limiti del sistema, Questi test, progettati per imitare scenari del mondo reale, hanno prodotto risultati assolutamente positivi, e ogni caso è stato superato con successo. Ciò non solo sottolinea l'adattabilità del sistema, ma conferma anche la sua affidabilità nella gestione di diverse query di ricerca. L'esecuzione riuscita sia della suite di test che dei casi di test aggiuntivi fornisce una forte prova dell'efficacia e della resilienza delle funzionalità avanzate di ricerca semantica e sintattica. Gli utenti possono fare affidamento sulla directory per fornire risultati accurati e pertinenti, migliorando l'esperienza utente complessiva e riaffermando la disponibilità del sistema per l'implementazione.

In conclusione, i risultati affermativi aprono la strada all'integrazione di queste funzionalità di ricerca avanzata nella Thing Description Directory, segnando una milestone significativa nell'evoluzione della piattaforma.

5.6 Problemi Riscontrati & Soluzioni

Come per qualsiasi progetto di sviluppo software, l'implementazione della Thing Description Directory WoTerFlow ha portato alla luce una serie di sfide uniche. Questo

capitolo approfondisce gli ostacoli affrontati durante il processo di sviluppo e le soluzioni strategiche implementate per superare queste sfide. Sono emersi due ostacoli principali: la complessa conversione tra JSON-LD ed RDF, e la necessità di un efficiente meccanismo di memorizzazione nella cache dei context, per ottimizzare le interazioni del server. In questa esplorazione sono descritte in dettaglio le complessità di ciascun problema, illustrando le soluzioni applicate per garantire il successo del progetto.

5.6.1 Conversione tra formati JSON-LD e RDF

La conversione tra JSON-LD e RDF ha rappresentato una sfida significativa a causa della necessità di implementare una logica precisa, che includesse l'utilizzo di funzionalità speciali di JSON-LD, tra cui l'**expansion** ed il **framing**, aggiungendo una interessante complessità al processo. Questa sezione esplora la profondità della sfida della conversione, svelando le complessità incontrate e le soluzioni strategiche ideate. al centro della sfida c'è la necessità di trasformare le strutture dati tra le rappresentazioni JSON-LD e RDF, ciascuna parte integrante della natura dinamica del WoT. Le caratteristiche uniche di JSON-LD, come la dipendenza dal contesto per la comprensione semantica, hanno richiesto un'implementazione meticolosa per garantire la fedeltà dei dati nei diversi formati.

Per risolvere questo problema è stato adottato un duplice approccio, sfruttando sia le librerie esistenti che le strategie di implementazione personalizzate. Le librerie utilizzate hanno svolto un ruolo fondamentale nel semplificare il processo di conversione. Ad esempio, considerando il seguente frammento che illustra l'uso di Apache Jena per convertire JSON-LD in RDF

```kotlin
fun toRdf(jsonLdString: String): Model {
    try {
        val document = JsonDocument.of(jsonLdString.reader())

        val expanded = JsonLd.expand(document)//.options(options11)
        val expandedDocument = expanded.get()

        val model = ModelFactory.createDefaultModel()

        expandedDocument.forEach {
            RDFDataMgr.read(model, it.toString().reader(), null, Lang.JSONLD11)
        }

        return model
    } catch (e: JsonLdError) {
        throw ThingException("${e.message}")
    } catch (e: Exception) {
        throw ThingException(
                        "The thing must contain a @context field: ${e.message}")
    }
}
```

Listing 43: Codice per conversione da JSON-LD a RDF

Nel codice mostrato si effettua prima un **expand** del documento sfruttando la libreria Apicatalog JsonLd. Successivamente viene effettuato un ciclo su ogni componente del documento espanso, utilizzando le api di Apache Jena per effettuare il read del modello specificando come linguaggio JSONLD11.

Di seguito viene presentato il codice per effettuare l'operazione contraria, convertendo da RDF/JSONLD11 a JSON-LD.

```
fun fromRdf(rdfModel: Model): ObjectNode {
    try {
        val document = RdfDocument.of(toString(rdfModel).reader())
        val jsonArray = JsonLd.fromRdf(document).options(options11).get()
        val jsonDocument = JsonDocument.of(jsonArray.toString().reader())

        val expanded = JsonLd.expand(jsonDocument).options(options11).get()
        val expandedDocument = JsonDocument.of(expanded)

        val framed = JsonLd.frame(expandedDocument, contextV11Document)
            .options(options11).get()
        val graphed = getGraphFromModel(framed)

        return graphed!!
    } catch (e: Exception) {
        throw ConversionException(
                        "Error converting from RDF to JSON-LD: ${e.message}")
    }
}
```

Listing 44: Codice per conversione da RDF a JSON-LD

In questo caso è stato possibile sfruttare le api della libreria Apicatalog JsonLd per effettuare la conversione. Viene letto il documento RDF ed è successivamente ottenuta una lista contenente tutte le componenti di tale. Possiamo quindi effettuare l'expand per estrarre tutti gli IRI. Viene poi effettuato il frame per estrarre esclusivamente la Thing Description, per ignorare tutti i contenuti del context esseno superflui avendo a disposizione il link di riferimento ai medesimi dati, non più necessari (onde ridurre il carico di memoria, potendo effettuare il retrieve in caso di necessità). In fine viene restituito il grafo contenente la struttura JSON-LD.

Questo approccio ha permesso una implementazione semplice ma efficace, rendendo possibile la conversione tra i due formati utilizzati dal sistema.

5.6.2 Caricamento in Cache dei Context

Nel campo delle Thing Description Directory, la gestione efficiente e il recupero dei context svolgono un ruolo fondamentale nel migliorare la reattività del sistema e alleviare potenziali bottleneck. Un aspetto cruciale è l'ottimizzazione del caricamento dei context, in particolare il recupero di documenti contestuali da server esterni. Per affrontare la sfida del recupero ripetuto di context online, è stato implementato un meccanismo in particolare, sfruttando il caricamento dei context nella cache. Questa strategia mira a

ridurre al minimo il carico del server, memorizzando nella cache locale i documenti di contesto a cui si è avuto accesso in precedenza, mitigando così il rischio di flooding dei server e potenziali blocchi. Questa misura è particolarmente cruciale, poiché affronta le sfide incontrate durante le fasi di test, in cui richieste eccessive hanno portato a restrizioni temporanee sull'accesso ai server interrogati per ottenere i dati dei context.

Il caricamento dei context nella cache funge da risposta strategica alle sfide affrontate durante la fase di test. Il meccanismo mira a migliorare l'efficienza del recupero del contesto, tutelando allo stesso tempo da potenziali problemi derivanti da restrizioni lato server dovute ad un numero eccessivo di richieste. Mettendo nella cache locale i documenti dei context a cui si è già avuto accesso in precedenza, questo approccio riduce al minimo la necessità di frequenti query online, riducendo il carico sui server esterni, eludendo potenziali blocchi, oltre che ridurre il traffico sulla rete stessa, velocizzando i tempi di esecuzione e di risposta.

Dettagli sull'Implementazione

Per consentire il caricamento dei context nella cache, è stata introdotta una classe ***CachedDocumentLoader***. Questa classe estende le funzionalità del caricatore di documenti standard e utilizza una cache basata su hashmap per archiviare gli URL e i corrispondenti documenti di contesto JSON-LD. Il metodo principale, loadDocument, controlla se un documento richiesto è presente nella cache. Se trovato, il documento viene recuperato dalla cache; in caso contrario viene sollevata un'eccezione che segnala l'assenza del documento nella cache locale

```
class CachedDocumentLoader(private val documentCache: Map<String, String>):
    DocumentLoader{

    override fun loadDocument(url: URI, options: DocumentLoaderOptions): Document {
        return if (documentCache.containsKey(url.toString())) {
            val cachedDocument = documentCache[url.toString()]!!
            val document = JsonDocument.of(cachedDocument.reader())

            document
        } else {
            throw DocumentLoaderException("Document not found in cache: $url")
        }
    }
}
```

Listing 45: Caricamento context in cache pt.1

Integrazione con RDFConverter

All'interno della classe RDFConverter, l'integrazione del caricamento dei context nella cache è vitale per mitigare i problemi sopra elencati. La hashmap ***documentCache*** viene creata per archiviare gli URL e i documenti contestuali associati. Durante l'inizializzazione, il documento di context rilevante viene scaricato e aggiunto alla cache. L'istanza option11 di JsonLdOptions è configurata con il CacheDocumentLoader personalizzato, abilitando in modo efficace il caricamento dei contesti nella cache per le operazioni successive.

```
class RDFConverter {
    ...
    val contextV11data: String = Utils.downloadFileAsString(contextV11)
    var contextV11Document: JsonDocument = JsonDocument.of(contextV11data.reader())

    val options11 = JsonLdOptions()

    init {
        val documentCache = mutableMapOf<String, String>()
        documentCache[contextV11] = contextV11data

        options11.documentLoader = CachedDocumentLoader(documentCache)
        options11.base = URI(contextV11)
        options11.processingMode = JsonLdVersion.V1_1
        ...
    }
    ...
}
```

Listing 46: Caricamento context in cache pt.2

5.6.3 Conclusioni

Il caricamento dei context sulla cache, nato dalla necessità di affrontare le sfide incontrate durante i test, emerge come una strategia cruciale nell'implementazione di una Thing Description Directory. Questo meccanismo non solo ottimizza il processo di recupero ma funge anche da misura proattiva per prevenire interruzioni causate da blocchi del server derivanti da un numero eccessivo di richieste verso i server interessati. Mettendo nella cache i context, l'implementazione di WoTerFlow diventa più resiliente, garantendo un'interazione più fluida con i server esterni

5.7 Funzionalità Supportate

In questa sezione, viene fornita una panoramica delle funzionalità supportate nell'implementazione dell Thing Description Directory WoTerFlow. L'implementazione aderisce alle specifiche delineate dallo standard del W3C, il quale specifica le diverse funzionalità essenziali, opzionali e raccomandate per la realizzazione di una Thing Directory che sia aderente a tali standard. È quindi mostrata una tabella dettagliata con le funzionalità essenziali (od obbligatorie), opzionali (o facoltative) e quelle raccomandate. Questa panoramica mira a fornire approfondimenti sulle funzionalità e sulle capacità del sistema implementato, evidenziando la misura in cui si allinea con le specifiche definite.

Feature	**Importanza**	**Stato Implementazione**
Things API	API Obbligatoria	Completa
Creation	Obbligatoria	Implementata
Retrieval	Obbligatoria	Implementata
Update	Obbligatoria	Implementata
Deletion	Obbligatoria	Implementata
Listing	Obbligatoria	Implementata
Validation	Raccomandata	Implementata
Events API	API Opzionale	Parziale
Creation	Opzionale	Implementata
Update	Opzionale	Implementata
Deletion	Opzionale	Implementata
All (notifica tutti gli eventi)	Opzionale	Implementata
Diff (differenze della modifica)	Opzionale	Implementazione Futura
Search API	API Opzionale	Completa
Semantic Search: SPARQL	Opzionale	Implementata
Syntactic Search: JSONPath	Opzionale	Implementata
Syntactic Search: XPath	Opzionale	Implementata

Tabella 5.4: Funzionalità TDD Supportate

API del servizio di Directory

Le API per il servizio Directory, per le Things e per la loro validazione sono state completamente implementate. Ciò include funzionalità come la creazione, il recupero, l'aggiornamento, l'eliminazione e l'elenco delle descrizioni delle cose all'interno della directory.

La validazione, sebbene fosse una feature raccomandata, è stata integrata nell'implementazione, in maniera tale da garantire che le descrizioni delle cose archiviate nella directory aderiscano allo schema e alla struttura definiti.

API degli Eventi

Le Event API, una funzionalità opzionale, sono state implementate in misura significativa. Sono difatti supportate le notifiche degli eventi come la creazione, l'aggiornamento e l'eliminazione delle Thing Description. Tuttavia, la funzionalità specifica per il calcolo delle differenze (diff) tra eventi non è stata ancora implementata.

API di ricerca avanzata

Le API di ricerca avanzata, un'altra funzionalità opzionale, sono state implementate per supportare varie modalità di ricerca. Ciò include le ricerche sintattiche tramite JSONPath e XPath, nonché ricerca semantica utilizzando query SPARQL. Queste funzionalità, oltre ad essere di particolare interesse per il progetto, migliorano la ricerca delle Thing Description all'interno della directory.

Conclusioni

Complessivamente, attraverso un'analisi esauriente delle funzionalità obbligatorie, facoltative e consigliate, è stato delineato come l'implementazione abbia soddisfatto pienamente le API per il servizio Directory, garantendo un'efficiente gestione delle Thing Description. Inoltre, l'integrazione della validazione, ha rafforzato la robustezza del sistema, promuovendo affidabilità e coerenza dei dati.
Tuttavia, nonostante il notevole progresso raggiunto, bisogna sottolineare che alcune funzionalità, come il calcolo delle differenze tra eventi, richiedono ulteriori sviluppi per essere pienamente implementate. Nonostante queste sfide, la panoramica delle capacità del sistema evidenzia la sua coerenza con le specifiche stabilite e suggerisce una solida base per sviluppi ed ottimizzazioni future. In definitiva, questa implementazione offre una solida infrastruttura per la gestione delle Thing Description, contribuendo a promuovere l'interoperabilità e l'efficienza nell'ecosistema del Web of Things.

Capitolo 6

Sviluppi Futuri

Sebbene l'attuale implementazione della Thing Description Directory abbia integrato con successo le funzionalità richieste dal W3C, le funzionalità avanzate come ricerca semantica e sintattica, nonché il servizio di notifica tramite Server-Sent Events (SSE), esistono diverse strade per ulteriori miglioramenti e personalizzazioni. Di seguito sono presentate alcune possibili direzioni di sviluppo futuro.

6.1 Configurazioni Personalizzate

Ignorare l'utilizzo dell'hashmap

Nell'implementazione attuale, viene utilizzata una hashmap per le operazioni di ricerca efficienti relative alla ricerca per ID, il retrieve generale e operazioni di ricerca tramite query JSONPath e XPath. Un miglioramento futuro potrebbe comportare l'introduzione di configurazioni personalizzate che consentono agli utenti di alternare l'utilizzo dell'hashmap per le operazioni di ricerca o di ignorarlo completamente. Questa flessibilità potrebbe soddisfare casi d'uso in cui gli utenti potrebbero dare priorità a diversi aspetti come l'efficienza della memoria.

6.2 Tecniche di ottimizzazione delle query

Meccanismi di memorizzazione nella cache

Si potrebbero integrare meccanismi di memorizzazione nella cache per archiviare le query eseguite di frequente e i relativi risultati. In questo modo, il sistema potrebbe ridurre il tempo di esecuzione delle informazioni comunemente richieste, migliorando la reattività complessiva della Thing Description Directory

6.3 Meccanismi di notifica estensibili

Integrazione con Protocolli di Notifica Aggiuntivi

Nonostante il requisito per la notifica degli eventi sia l'utilizzo del protocollo SSE, molti sistemi IoT moderni utilizzano altri protocolli. Il supporto a protocolli come MQTT o AMQP potrebbe ampliare la portata della comunicazione in tempo reale e della propagazione degli eventi all'interno dell'ecosistema WoT.

Con l'evoluzione del campo dell'IoT e del WoT, questi lavori futuri proposti mirano a posizionare la Thing Description Directory come una soluzione versatile e adattiva, in grado di soddisfare perfettamente i requisiti emergenti e i progressi tecnologici. Sia gli utenti che gli sviluppatori trarrebbero vantaggio da una TDD estensibile e personalizzabile, consentendo loro di adattare il sistema alle loro esigente e preferenze specifiche.

Capitolo 7

Conclusioni

In questa esplorazione dell'implementazione di una Thing Description Directory, sono state integrate diverse funzionalità e tecnologie chiave per creare una soluzione solida e versatile. La fusione di Server-Sent Events (SSE), il linguaggio di programmazione Kotlin con il suo costrutto Flow, la ricerca sintattica tramite JSONPath e XPath, la ricerca semantica tramite SPARQL e TDB incorporato tramite Apache Jena ha prodotto una piattaforma completa che affronta le complessità della gestione e delle complessità della gestione e dell'uso di query per le Thing Description nell'ecosistema del Web of Things.

Scelte tecnologiche e innovazioni

L'utilizzo dei benchmark, che hanno messo a confronto i linguaggi C#, Java e Kotlin, ha fornito preziose informazioni sui punti di forza e sui limiti di ciascun linguaggio. La scelta di Kotlin è giustificata non solo dalle sue prestazioni ma anche dalla sua sintassi concisa, dalla sua potenza espressiva e dalla perfetta integrazione con le librerie Java esistenti.

La decisione di sfruttare Kotlin, con la sua perfetta integrazione della programmazione asincrona tramite i Flow, si è rivelata determinante nella gestione della comunicazione in tempo reale tramite SSE. Ciò non solo migliora la reattività del sistema, ma si allinea anche ai moderni paradigmi della programmazione reattiva.

La funzionalità di ricerca sintattica che utilizzano XPath e JSONPath forniscono agli utenti potenti strumenti per individuare con precisione le informazioni all'interno delle Thing Description, mentre la ricerca semantica tramite SPARQL aggiunge un livello di comprensione semantica, arricchendo le capacità complessive delle query.

L'integrazione di TDB incorporato tramite Apache Jena garantisce un eseguibile autonomo, semplificando la distribuzione ed eliminando la necessità di installazioni esterne.

Questa scelta è in linea con i vincoli stabiliti, enfatizzando un approccio minimalista alle dipendenze e alla facilità d'uso.

L'uso innovativo di una hashmap per operazioni di ricerca dati efficienti introduce un elemento di ottimizzazione nel sistema, dimostrando un impegno per prestazioni e reattività.

Standardizzazione e Compatibilità

Il supporto dei formati JSON-LD e RDF garantisce l'interoperabilità, consentendo un'integrazione perfetta con una vasta gamma di dispositivi e piattaforme IoT. L'utilizzo di una suite per l'integration testing, approvata dal working group ufficiale, garantisce l'affidabilità e la stabilità della Thing Description Directory, garantendo agli utenti la sua robustezza in diversi scenari di implementazione.

In conclusione, la Thing Description Directory WoTerFlow costituisce una testimonianza della riuscita integrazione di tecnologie all'avanguardia ed approcci innovativi. La sezione dei lavori futuri ha delineato potenziali percorsi di miglioramento, comprese tecniche avanzate di ottimizzazione delle query, supporto esteso per le query, e possibilità di personalizzazione dell'applicativo.

Mentre l'IoT ed in particolare il WoT continuano ad evolversi, la Thing Description Directory presentata si posiziona come una soluzione dinamica e adattiva, pronta a soddisfare le sfide ed i requisiti emergenti del panorama IoT. Abbracciando una filosofia di progettazione performante, WoTerFlow pone le basi per un ecosistema IoT in cui efficienza, flessibilità e innovazione si incontrano per dare forma alla prossima frontiera dei sistemi connessi.

Bibliografia

[1] Web of Things - W3C
https://www.w3.org/TR/wot-discovery/#

[2] W3C
https://www.w3.org/

[3] TinyIoT
https://github.com/TinyIoT/thing-directory

[4] WotHive
https://github.com/oeg-upm/wot-hive

[5] Zion
https://zion.vaimee.com/

[6] GitHub
https://github.com/

[7] GoLang
https://go.dev/

[8] RDF
https://www.w3.org/RDF/

[9] Triple Store - Wikipedia
https://en.wikipedia.org/wiki/Triplestore

[10] SPARQL
https://www.w3.org/TR/sparql11-query/

[11] XPath
https://www.w3.org/TR/xpath-31/

[12] JSONPath
https://datatracker.ietf.org/doc/html/draft-ietf-jsonpath-base

[13] Docker
https://www.docker.com/

[14] LevelDB
https://github.com/google/leveldb

[15] Google
https://www.google.com/

[16] Java
https://www.java.com/en/

[17] Server-Sent Events (SSE)
https://www.w3.org/TR/2009/WD-eventsource-20091029/

[18] Java Virtual Machine (JVM) - Wikipedia
https://en.wikipedia.org/wiki/Java_virtual_machine

[19] C#
https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

[20] Kotlin
https://kotlinlang.org/

[21] Microsoft - Wikipedia
https://en.wikipedia.org/wiki/Microsoft

[22] Microsoft .NET
https://dotnet.microsoft.com/en-us/

[23] C++
https://learn.microsoft.com/cpp/dotnet/dotnet-programming-with-cpp-cli-visual-cpp

[24] Sun Microsystems - Wikipedia
https://en.wikipedia.org/wiki/Sun_Microsystems

[25] JetBrains
https://www.jetbrains.com/

[26] Benchmark
https://programming-language-benchmarks.vercel.app/

[27] Benchmark: Java-C#
https://programming-language-benchmarks.vercel.app/java-vs-csharp

[28] Benchmark: Java-Kotlin
https://programming-language-benchmarks.vercel.app/java-vs-kotlin

[29] Benchmark: C#-Kotlin
https://programming-language-benchmarks.vercel.app/csharp-vs-kotlin

[30] JSON-LD - W3C
https://www.w3.org/TR/json-ld11/

[31] Ktor
https://ktor.io/

[32] Ktor: CIO
https://ktor.io/docs/http-client-engines.html#cio

[33] Apache Jena
https://jena.apache.org/

[34] OWL Visualization
https://www.w3.org/2018/09/rdf-data-viz/

[35] Jena Architecture
https://jena.apache.org/about_jena/architecture.html

[36] Icons8
http://icons8.com/icons

[37] FasterXML Jackson
https://github.com/FasterXML/jackson

[38] Apicatalog Titanium
https://github.com/filip26/titanium-json-ld

[39] XML Validation
https://www.w3.org/ns/shacl.rdf

[40] TTL Validation
https://github.com/w3c/wot-thing-description/blob/main/validation/
td-validation.ttl

[41] Postman
https://www.postman.com/

[42] JayWay JsonPath
https://github.com/json-path/JsonPath

[43] Saxon HE
https://github.com/Saxonica/Saxon-HE

[44] Farshidtz wot-discovery-testing
https://github.com/farshidtz/wot-discovery-testing/tree/main/directory

[45] W3C wot-testing template
https://github.com/w3c/wot-discovery/tree/main/testing