Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

Analisi ed Estensione di una Thing Description Directory

Relatore:
Chiar.mo Prof.
Davide Rossi

Presentata da:
Nicolas Cola

Sessione II
Anno Accademico 2024/2025

Indice

Capitolo 1

Introduzione

Il panorama tecnologico odierno è caratterizzato da una crescita esponenziale nell'interconnessione dei dispositivi che caratterizzano la nostra quotidianità, fenomeno definito Internet of Things (IoT). Questo si manifesta attraverso l'integrazione di dispositivi intelligenti e attivi che comunicano tra loro: sistemi di automazione domestica, veicoli dotati di comunicazione inter-veicolare, sensori ambientali.

Ciò ha fatto emergere molti problemi relativi all'interoperabilità e alla standardizzazione della comunicazione inter-dispositivo, il che ha portato alla necessità di stabilire modalità comuni di comunicazione tra dispositivi che operano in domini diversi.

Il paradigma per affrontare questi problemi è il Web of Things (WoT) [1]. Attraverso l'adozione dei principi architetturali e delle tecnologie consolidate del World Wide Web, fornisce un framework uniforme per l'integrazione e l'accesso ai dispositivi, superando le barriere di interoperabilità dell'IoT.

Il W3C ha formalizzato questo paradigma utilizzando tecniche fondamentali che costituiscono l'architettura WoT. La Thing Description (TD) [3] definisce un formato standardizzato basato su JSON-LD [4] per descrivere le capacità, le interfacce e i metadati di qualsiasi dispositivo IoT in modo machine-readable. Un'altra specifica è WoT Discovery [2], la quale definisce meccanismi di scoperta dei dispositivi registrati in rete.

WoTerFlow [8] nasce come un'implementazione di una Thing Description Directory (TDD) conforme alle specifiche del W3C. Una TDD è un registro o catalogo che memorizza e fornisce accesso alle Thing Description di più dispositivi. Funge da repository consultabile in cui i dispositivi IoT possono registrare le proprie TD e i client possono scoprire e ricercare i dispositivi disponibili e le loro funzionalità.

1.1 Obiettivi del progetto

Il presente lavoro si propone di migliorare ed estendere le funzionalità di WoTerFlow attraverso i seguenti obiettivi specifici:

- **Risoluzione delle problematiche esistenti**: analisi e correzione dei bug e delle limitazioni presenti nell'implementazione corrente
- **Estensione delle funzionalità**: sviluppo di nuove caratteristiche per migliorare l'usabilità e le prestazioni del sistema
- **Miglioramento dell'architettura**: ottimizzazione della struttura del codice per garantire maggiore manutenibilità e scalabilità

1.2 Struttura del documento

Il capitolo 2 presenta una panoramica generale del Web of Things e dei suoi componenti fondamentali. Nel capitolo 3 sarà presentata una panoramica delle tecnologie utilizzate e delle funzionalità di WoTerFlow. Successivamente, nel capitolo 4, saranno esposte nel dettaglio le modifiche e le estensioni implementate, con particolare attenzione agli aspetti tecnici e alle scelte progettuali adottate.

Capitolo 2

Web of Things

Il **Web Semantico**, anche noto come **Web 3.0**, rappresenta un'estensione del World Wide Web in cui le informazioni non sono soltanto leggibili dall'uomo, ma anche dalle macchine. L'obiettivo è favorire l'interoperabilità tra applicazioni appartenenti a domini differenti, eliminando ambiguità nell'interpretazione dei dati, rendendoli facilmente leggibili dalle macchine.

Nonostante la rapida crescita dell'**Internet of Things (IoT)**, l'interoperabilità continua a essere un problema importante. Dispositivi forniti da diversi fornitori spesso funzionano da soli, utilizzando protocolli proprietari e piattaforme chiuse. Ciò crea ecosistemi frammentati che impediscono l'integrazione e l'innovazione. Il processo di individuazione, identificazione e accesso ai dispositivi in rete è una delle sfide principali. Infatti, un dispositivo IoT deve poter essere scoperto e descritto in modo standardizzato in modo che tutti possano interpretarlo.

Per affrontare queste sfide, il **Web of Things (WoT)**, sviluppato dal World Wide Web Consortium (W3C), propone un approccio basato su tecnologie web e principi semantici già affermati. Esso introduce un modello architetturale che consente di rappresentare dispositivi e servizi IoT come risorse Web interoperabili.

Pertanto, WoT può essere considerato un'implementazione concreta del Web Semantico nell'ecosistema Internet of Things. Ciò consente la scoperta, la comprensione e l'integrazione automatica dei dispositivi da parte di applicazioni appartenenti a diversi domini, consentendo una completa interoperabilità semantica tra il mondo digitale e quello fisico.

2.1 Thing Description

La **Thing Description (TD)** è un elemento fondamentale nel WoT del W3C e può essere considerata il punto di ingresso a un dispositivo. Si tratta di un documento che utilizza metadati strutturati per descrivere le capacità di un dispositivo e le modalità di interazione che esso espone sulla rete.

Le TD sono rappresentate in JSON-LD (JavaScript Object Notation for Linked Data), formato standard per la rappresentazione delle Thing Description nel Web of Things. Questo permette di incorporare dati strutturati linked data utilizzando la sintassi JSON, creando così una struttura di dati interconnessi. Le proprietà caratterizzanti di un documento JSON-LD per Thing Description includono:

- **@context**: definisce i vocabolari semantici utilizzati (tipicamente il WoT TD context)
- **@type** che specifica il tipo di risorsa ("Thing")
- **@id** per l'identificatore univoco del dispositivo.
- **Interaction Affordances**: proprietà che definiscono le modalità di interazione con il dispositivo. Permettono di specificare il protocollo utilizzato e i formati di dati utilizzati dal dispositivo, evitando disambiguità ed errori di interpretazione. Le principali sono:
  - **properties**: attributi leggibili/scrivibili del dispositivo (es. acceso/spento)
  - **actions**: operazioni invocabili sul dispositivo (es. accendi/spegni)
  - **events**: una sorgente di eventi, che invia in modo asincrono i dati degli eventi ai subscriber dell'evento (es. avvisi di surriscaldamento)

  .
- **securityDefinitions e security**: metadati sui meccanismi di sicurezza che devono essere utilizzati per le interazioni.
- **links**: collegamenti per esprimere qualsiasi relazione formale o informale con altri dispositivi o documenti sul Web.

```
{
    "@context": "https://www.w3.org/2022/wot/td/v1.1",
    "id": "urn:dev:ops:32473-WoTLamp-1234",
    "title": "MyLampThing",
```

```
    "securityDefinitions": {
        "basic_sc": {"scheme": "basic", "in":"header"}
    },
    "security": ["basic_sc"],
    "properties": {
        "status" : {
            "type": "string",
            "forms": [{
                "href": "https://mylamp.example.com/status",
                "htv:methodName":"GET"
            }]
        }
    },
    "actions": {
        "toggle" : {
            "forms": [{
                "href": "https://mylamp.example.com/toggle",
                "htv:methodName":"POST"
            }]
        }
    },
    "events":{
        "overheating":{
            "data": {"type": "string"},
            "forms": [{
                "href": "https://mylamp.example.com/oh",
                "htv:methodName":"GET",
                "subprotocol": "longpoll"
            }]
        }
    }
}
```

Listing 1: Thing Description di una lampada

Nel complesso, la Thing Description promuove l'interoperabilità in due modi: in primo luogo, le TD consentono la comunicazione macchina-macchina nel Web of Things. In secondo luogo, i TD possono fungere da formato comune e uniforme per

gli sviluppatori per documentare e recuperare tutti i dettagli necessari per creare applicazioni in grado di accedere ai dispositivi IoT e utilizzarne i dati.

2.2 WoT Discovery

Per poter interagire con una Thing, è necessario prima individuare e ottenere la relativa Thing Description. A questo scopo interviene il processo di **WoT Discovery**, che definisce i meccanismi attraverso cui le Things possono essere scoperte, identificate e descritte in ambienti IoT eterogenei, garantendo interoperabilità e accesso uniforme alle risorse. Il processo di scoperta deve integrarsi con i meccanismi esistenti, garantendo sicurezza, tutela della privacy e aggiornamento efficiente delle TD in un ecosistema IoT dinamico e diversificato.

Per gestire questi requisiti, il processo di Discovery si articola in due fasi principali: Introduzione ed Esplorazione.

2.2.1 Introduzione

La fase di **Introduzione** sfrutta meccanismi di scoperta esistenti, ma non espone direttamente i metadati: restituisce invece URL che puntano ai servizi di esplorazione. Questa separazione consente di mantenere protette le informazioni sensibili attraverso controlli di autenticazione e autorizzazione. I principali meccanismi di introduzione includono:

- **Direct**: qualsiasi meccanismo che si traduce in un singolo URL. Questo include beacon Bluetooth, codici QR e URL scritti dall'utente.
- **Well-Known URIs**: registrando la Thing Description a `/.well-known/wot`. Quando viene effettuata una richiesta alWell-Known URI precedente, il server restituirà la TD.
- **mDNS (Multicast DNS)**: meccanismo di rilevamento della rete locale che consente ai dispositivi di rilevare e pubblicizzare servizi sulla stessa rete locale senza richiedere un server DNS centralizzato.
- **CoRE Link Format e CoRE Resource Directory**: permette di registrare un link alla TD del dispositivo corrispondente.
- **DID Documents**: identificatori distribuiti o basati su blockchain che possono puntare alla posizione dei dispositivi d senza fare affidamento su autorità centralizzate

2.2.2 Esplorazione

La fase di **Esplorazione**, invece, permette alle parti autorizzate di accedere effettivamente alle TD. I servizi di esplorazione forniscono i metadati solo dopo aver applicato adeguati controlli di sicurezza. Esistono due principali tipologie di meccanismi di esplorazione: i Thing Description Servers (TD Servers) e le Thing Description Directories (TDD).

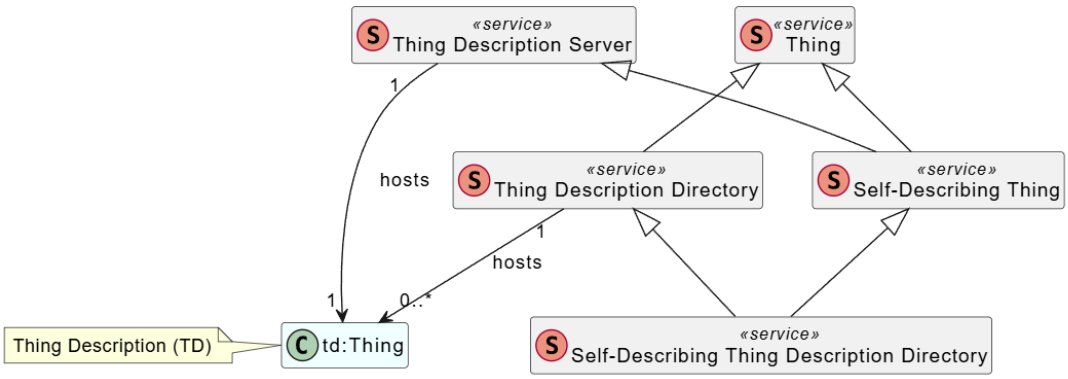

Figura 2.1: Diagramma di classe di alto livello dei meccanismi di esplorazione

La figura 2.1 illustra il modello informativo di alto livello per i Thing Description Servers e i servizi di Thing Description Directory.

I **TD Servers** sono un qualsiasi servizio web a cui è possibile fare riferimento tramite un URL e che restituisce una TD dopo controlli di autenticazione e accesso appropriati. Un utilizzo tipico è per implementare una funzionalità di autodescrizione di un dispositivo. Ciò avviene facendo sì che la Thing ospiti la propria Thing Description rendendola disponibile tramite una risorsa web identificata con un URL.

Una **TDD**, invece, può contenere TD e allo stesso tempo essere anche una Thing, il che significa che è dotata di una propria descrizione. Essa fornisce endpoint di servizi web per la registrazione, la modifica, il recupero e la ricerca di Thing Description.

2.3 Binding Templates

L'IoT utilizza una varietà di protocolli per l'accesso ai dispositivi, poiché nessun protocollo è appropriato in tutti i contesti. Pertanto, una sfida centrale per il Web of Things è consentire interazioni con il vasto numero di diverse piattaforme IoT e dispositivi che non seguono alcuno standard particolare, ma forniscono un'interfaccia idonea su un determinato protocollo di rete. I **Binding Templates**[5] consentono

a un client di utilizzare la TD per estrarre i metadati specifici dei protocolli (ad esempio, HTTP, MQTT ecc.), dei formati del payload (binario, JSON ecc.) e del loro utilizzo nel contesto di una piattaforma IoT. I metadati estratti possono essere trasmessi a un'interfaccia di implementazione di rete per stabilire l'interazione con il dispositivo e serializzare/deserializzare i messaggi del payload. In tale contesto, i Binding Templates includono tre tipi di binding:

- **Protocol Bindings**: sono la mappatura di un'Interaction Affordance a messaggi concreti di un protocollo specifico come HTTP, CoAP o MQTT. Informa il Consumer su come attivare l'Interaction Affordance tramite un'interfaccia di rete.
- **Format Bindings**: definiscono i formati di rappresentazione utilizzati per lo scambio di dati. Possono variare a seconda dei protocolli, delle piattaforme IoT e degli standard.
- **Platform Bindings**: definiscono il modo in cui un dispositivo si integra con una particolare piattaforma IoT o ecosistema. Ciò risulta di fondamentale importanza dal momento in cui i dispositivi IoT non sono isolati, ma spesso sono gestiti tramite piattaforme o ecosistemi specifici.

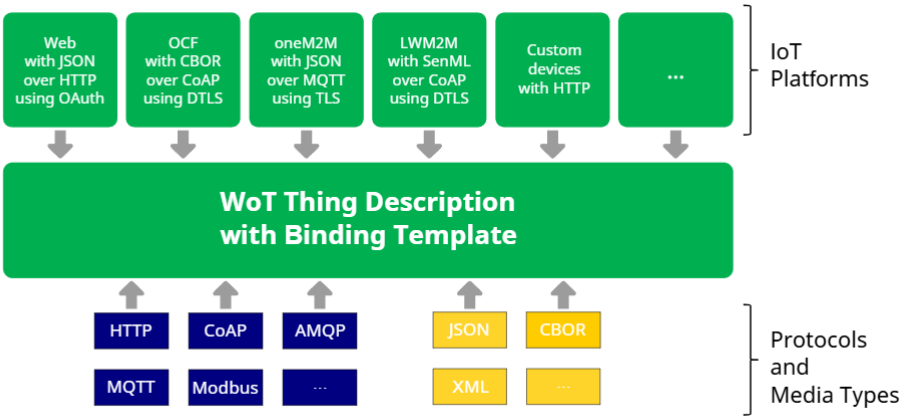

Figura 2.2: Utilizzo dei Binding Templates in una Thing Description

La figura 2.2 illustra come le TD con Binding Templates fungano da livello di astrazione unificato tra più piattaforme IoT (in alto) e i protocolli e formati di dati (in basso) che i dispositivi utilizzano per comunicare. Ciò consente a una TD di essere mappata su diverse combinazioni specifiche, senza dover implementare ciascuna di esse separatamente. In questo modo si favorisce l'interoperabilità tra diversi ecosi-

stemi IoT, mantenendo la chiarezza semantica su come diverse piattaforme possono accedere allo stesso dispositivo.

2.4 Servient

Nell'ecosistema del Web of Things si hanno dispositivi IoT con specifiche funzionalità e protocolli di comunicazione, Thing Description che li descrivono in modo standardizzato e applicazioni che interagiscono con essi. Dal momento in cui i dispositivi sono oggetti fisici, le applicazioni dei software e le TD dei semplici documenti, la sfida principale consiste nel determinare un modo per collegare tutti questi pezzi insieme.

Un **Servient**[6] permette di fare proprio questo. In particolare, un Servient è un agente software che gestisce una o più Thing, le espone in base alle loro TD e fornisce i meccanismi affinché altre applicazioni o agenti possano scoprire e interagire con esse.

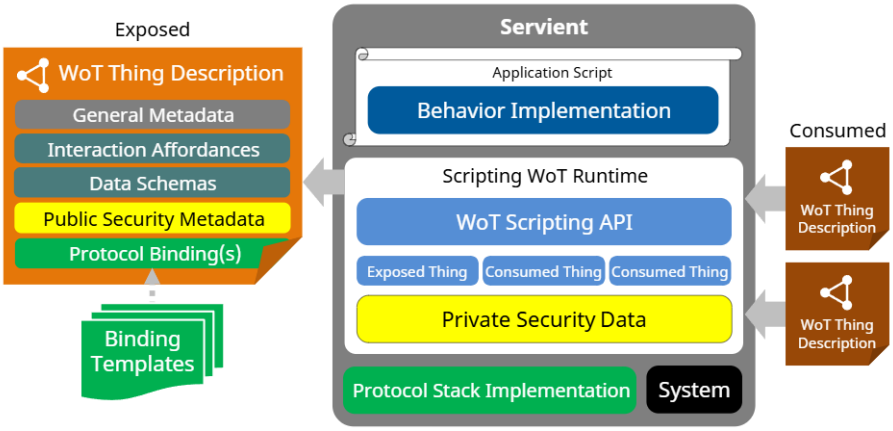

Figura 2.3: Struttura di un Servient

La figura 2.3 mostra nel dettaglio la struttura di un Servient. Sul lato sinistro, si vede la prospettiva Exposed, che mostra ciò che il Servient fornisce al mondo esterno. Questa rappresenta le TD che il Servient pubblica e rende disponibili. Sul lato destro, si vede la prospettiva Consumed, rappresenta le TD che legge e utilizza per interagire con i dispositivi esterni. Al centro, si trova il Servient stesso, che è l'agente attivo che collega queste due prospettive. Esso è composto da:

- **Application Script**: questo layer contiene il codice che implementa le funzionalità del dispositivo.
- **Scripting WoT Runtime**: contiene l'ambiente di esecuzione del comportamento del dispositivo. Qui risiede:
  - **Scripting API**: un'API JavaScript che fornisce agli sviluppatori un modo semplice e unificato per interagire con i dispositivi associati alle relative TD, nascondendo i dettagli implementativi e i protocolli utilizzati.
  - **Exposed Thing e Consumed Thing**: rappresentazioni software dei dispositivi basate sulle loro TD. L'astrazione Exposed Thing rappresenta un oggetto ospitato localmente e accessibile dall'esterno, consentendo al Servient di agire come fornitore, rendendo i dispositivi gestiti disponibili come parte dell'ecosistema IoT. Invece, l'astrazione Consumed Thing rappresenta un Thing ospitato in remoto per i Consumer a cui è necessario accedere tramite un protocollo di comunicazione. Essi permettono al Servient di agire come consumatore o client, consentendogli di integrarsi e coordinarsi con più dispositivi e servizi.
  - **Private Security Data**: informazioni sensibili che devono rimanere private (chiavi crittografiche, password ecc.).
- **Protocol Stack Implementation**: layer che implementa i protocolli concreti (HTTP, MQTT, CoAP, ecc.) specificati nei Binding Templates.
- **System**: componente che gestisce le funzioni generali del sistema operativo.

2.5 Resource Description Framework

Il **Resource Description Framework** costituisce 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. Esso permette, nel contesto del WoT, la modellazione delle Things in modo flessibile, indicando chiaramente i concetti e le connessioni tra di esse.

Il modello dei dati di RDF è formato da tre elementi chiave:

- **Subject**: rappresenta l'entità principale (o la risorsa) su cui si stanno fornendo le informazioni.
- **Predicate**: 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.

- **Object**: rappresenta il valore o la risorsa associata ad un dato predicato nel contesto di riferimento. Questo può quindi essere un valore letterale, un valore numerico, un oggetto complesso o addirittura un'altra risorsa.

Questi tre componenti fondamentali costituiscono la cosiddetta **RDF Triple**. Un insieme di triple collegate tra loro forma un grafo diretto. Ciò 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 l'interoperabilità delle risorse digitali (e di conseguenza, della loro controparte fisica).

2.5.1 Named Graphs e Graph Literals

Nel contesto di una Thing Description Directory, un aspetto fondamentale per la gestione efficace delle TD è la distinzione delle strutture supportate da RDF per il raggruppamento delle triple: i Named Graphs e i Graph Literals.

Un **Named Graph** è un insieme di triple RDF identificato da un URI univoco. Ogni tripla all'interno di un Named Graph è associata a questo identificatore, permettendo di organizzare e gestire separatamente insiemi di informazioni correlate. Nel contesto di una TDD, ogni Thing Description può essere rappresentata come un Named Graph distinto, identificato dall'URI della Thing stessa.

Al contrario, i **Graph Literals** rappresentano un grafo RDF incapsulato come valore letterale all'interno di una tripla. A differenza dei Named Graphs, sono immutabili e vengono trattati come valori atomici.

Poiché le Thing Descriptions sono generalmente soggette a modifiche, l'utilizzo di Named Graphs è l'approccio migliore per organizzare le TD in una Directory. Questo permette una gestione flessibile e scalabile delle risorse.

2.5.2 Triple Store

All'interno di una Thing Description Directory sono registrate multiple TD. Un elemento di fondamentale importanza per gestirle al meglio è il **Triple Store**. Un Triple Store è un database progettato appositamente per l'archiviazione e il recupero di triple RDF. Utilizza 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 la creazione e gestione di strutture dati intricate e interconnesse e una rapida navigazione per il recupero di informazioni. In particolare, il supporto per Named Graphs nel Triple Store consente query efficienti sulle Thing Descriptions, facilitando operazioni come la scoperta, il filtraggio e la gestione del ciclo di vita delle TD.

Capitolo 3

WoTerFlow

WoTerFlow nasce come implementazione sperimentale di una Thing Description Directory (TDD), elemento fondamentale nell'architettura del Web of Things.
L'obiettivo di WoTerFlow è quello di fungere da registro centralizzato per la gestione di Thing Descriptions, con particolare attenzione allo sviluppo di funzionalità di ricerca semantica avanzata che consentano l'identificazione di dispositivi compatibili a parametri specifici.

3.1 Tecnologie utilizzate

L'implementazione efficace di un TDD implica decisioni critiche riguardo l'architettura di storage, i meccanismi di discovery, i protocolli di comunicazione e le tecnologie di elaborazione semantica. Di seguito saranno presentate le principali tecnologie che compongono WoTerFlow, con particolare attenzione alle motivazioni che hanno portato alla loro adozione.

3.1.1 Linguaggio

WoTerFlow è sviluppato in Kotlin[9], linguaggio scelto per l'ampia disponibilità di librerie per tecnologie semantiche, fondamentali nel Web of Things.

3.1.2 Server

Il server è implementato usando il framework Ktor[10] che consente di sfruttare appieno le potenzialità del linguaggio Kotlin, grazie al suo supporto nativo per la programmazione asincrona guidata dalle coroutine di Kotlin. Un importante vantaggio di Ktor è la possibilità di selezionare l'engine, ovvero il software che gestisce

come il server processa le richieste e invia le risposte agli utenti, che più rispecchia le caratteristiche ricercate per il proprio server.
L'engine utilizzato per WoTerFlow è il Coroutine-based I/O (CIO)[11], che pone l'enfasi sull'asincronicità delle coroutine di Kotlin e la leggerezza in termini di risorse. Ciò lo rende la scelta ideale per le applicazioni eseguite su dispositivi IoT con potenza di elaborazione e memoria limitate, senza inficiare sulle prestazioni.

3.1.3 Persistenza dei dati

Le Thing Description, essendo documenti semantici strutturati, sono naturalmente rappresentabili in formato RDF. Di conseguenza, per la persistenza delle Thing Description, WoTerFlow si serve del framework Apache Jena[12], il quale mette a disposizione TDB2, un Triple Store progettato per la gestione efficiente di grafi RDF, facilitando notevolmente le operazioni di discovery. Infatti, grazie al supporto di Apache Jena per il linguaggio di query semantico SPARQL[13], permette di recuperare e manipolare le Thing Description in modo efficiente.

Inoltre, in modo da ottimizzare le prestazioni durante le operazioni di ricerca, WoTerFlow implementa un meccanismo di caching. Una HashMap mantiene in memoria le Thing Description, permettendo così di recuperare le TD con complessità computazionale costante O(1), eliminando l'overhead che si avrebbe con gli accessi su disco dovuto dalle query SPARQL. Questa cache viene inizializzata, popolandola con le Thing Description memorizzate, all'avvio del sistema e mantenuta sincronizzata con TDB2. Questo è possibile facendo sì che ogni modifica al Triple Store comporti anche l'aggiornamento della cache, garantendo consistenza dei dati.

3.1.4 JSON-LD

Siccome le TD sono scritte in formato JSON-LD ma vengono salvate in formato RDF, WoTerFlow si avvale della libreria **Titanium JSON-LD**[15], utilizzata principalmente per effettuare le conversioni da JSON-LD a RDF e viceversa. Questa libreria permette il parsing, la manipolazione e serializzazione dei dati JSON-LD, rispettando le specifiche W3C per il formato JSON-LD. Le principali funzionalità supportate dalla libreria includono:

- **Context Handling**: supporta l'uso dei context di JSON-LD, permettendo di definire e gestire vocbolari interni ed esterni.
- **Operazioni Compact, Expand, Frame e Flatten**: 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**: supporta operazioni di conversione da RDF a JSON-LD e viceversa.

Infine, questa libreria gioca un ruolo fondamentale nella gestione dei metadati del WoT, offrendo delle solide fondamenta per encoding/decoding efficienti e manipolazione delle Thing Description.

3.1.5 Server-Sent Event e Flow

Server-Sent Events (SSE) costituisce una tecnologia per implementare comunicazioni unidirezionali in tempo reale dal server verso il client, particolarmente adatta per la notifica degli eventi previsti dalla Notification API, definita nella specifica WoT Discovery.

Le caratteristiche principali includono:

- **Connessioni persistenti**: mantengono il canale aperto per notifiche continue
- **Riconnessione automatica**: gestione automatica delle interruzioni di rete tramite tentativi di riconnessione in caso di disconnessione
- **Formato text/event-stream**: ogni evento è costituito da una serie di campi, tra cui il nome dell'evento, il payload e gli identificatori opzionali. Ciò consente di distinguere i diversi tipi di eventi

Dato che Kotlin non supporta nativamente SSE, WoTerFlow lo implementa facendo uso del costrutto MutableSharedFlow della libreria standard del linguaggio.
Si tratta di un costrutto per gestire stream di dati asincroni, risultando ideale per implementare funzionalità SSE nelle Thing Description Directory.

Le caratteristiche principali di MutableSharedFlow includono:

- **Asincronicità**: supporta la programmazione asincrona senza il blocco dei thread, consentendo una gestione efficiente di attività concorrenti.
- **Hot streams**: è un flusso caldo, il che significa che inizia a emettere valori indipendentemente dal fatto che vengano richiesti o meno. Ciò consente di pubblicare determinati eventi anche quando nessun subscriber è iscritto ad essi, garantendo che non vadano persi.
- **Iscrizione multipla**: più subscriber possono iscriversi allo stesso MutableSharedFlow, ricevendo gli stessi eventi.

3.2 Funzionalità

In questa sezione si pone l'attenzione sulle funzionalità e i dettagli implementativi di WoTerFlow. La specifica W3C Wot Discovery definisce le features che una Thing Description Directory deve obbligatoriamente implementare per esssere considerata tale, lasciando anche spazio alla personalizzazione a seconda delle esigenze di ciascun progetto. Infatti, oltre alle caratteristiche obbligatorie, la specifica ne presenta alcune opzionali e raccomandate, dando allo sviluppatore la flessibilità e facoltà di implementarle o meno.
La tabella 3.1 presenta le funzionalità dalla specifica W3C e mostra quali WoTerFlow implementa. Come si può notare, tutte le funzionalità obbligatorie e la maggioranza di quelle facoltative sono state implementate.

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

Tabella 3.1: Funzionalità TDD Supportate

Per quanto riguarda i dettagli implementativi, WoTerFlow è basato su un'architettura model-service-controller composta da:

- **Model**: è la struttura dati centrale dell'applicazione, in questo caso il Triple Store di Apache Jena. Il compito principale di tale layer è quello di gestire, interrogare e garantire la persistenza dei dati.

- **Controller**: responsabile di ricevere, validare e gestire le richieste HTTP. Esso delega la responsabilità al Service di eseguire le operazioni di business e, sulla base del risultato di queste, fornisce una risposta HTTP adeguata al Client.
- **Service**: esegue le operazioni di business delegate dal Controller, interagendo con il Model per raccogliere e aggiungere dati.

Pertanto, in WoTerFlow, ogni funzionalità è implementata avendo coppie di classi Controller-Service. Tale approccio porta numerosi benefici all'applicazione, i principali includono:

- **Separazione delle responsabilità**: ogni layer possiede un compito specifico e ben definito, facilitando la comprensione e la navigazione del codice.
- **Testabilità**: i componenti possono essere testati in isolamento, consentendo unit test più rapidi, mirati e affidabili.
- **Manutenibilità**: grazie al basso accoppiamento dei componenti e alla loro coesione, la modifica e l'estensione delle funzionalità risulta meno onerosa.

3.2.1 Things

La **Things API** è un'API HTTP RESTful servita all'endpoint `/things` che fornisce interfacce per creare, recuperare, aggiornare, eliminare ed elencare le TD.

Creazione

La creazione di una Thing Description si riferisce alla registrazione della stessa all'interno della directory. Il processo di creazione di una TD può avvenire in due modi:

1. **Creazione anonima**: quando la Thing Description inserita non ha un id, viene detta "anonima". Questa tipologia di creazione delega alla Directory l'assegnazione dell'id per consentire la gestione e il recupero della stessa. L'identificatore deve essere un UUID versione 4, un numero casuale o pseudo-casuale che non trasporta informazioni indesiderate sull'host o sulla TD.
2. **Creazione non anonima**: se la TD ha già un id, è sufficiente fare una richiesta PUT all'endpoint delle /things specificandone l'id nel path. Se la Thing Description non esisteva già verrà registrata.

Come descritto alla sezione 3.2.1, la thing description deve essere validata prima di essere inserita.

Recupero

Il recupero di una TD esistente deve essere effettuato utilizzando una richiesta HTTP GET all'endpoint /things/id, dove id è l'identificatore univoco della TD. In particolare, in WoTerFlow, questa funzionalità è implementata, come discusso in 3.1.3, con l'utilizzo di una MutableMap che funge da cache, aumentandone notevolmente l'efficienza.

Aggiornamento

Le operazioni di aggiornamento servono a sostituire o modificare parzialmente un TD esistente. In particolare, se l'obiettivo è solo aggiornare dei campi specifici, è sufficiente effettuare una richiesta HTTP PATCH a /things/id. Altrimenti, se si vuole sostituire completamente la TD, è necessario fare una richiesta PUT allo stesso endpoint.

Eliminazione

Un'operazione di eliminazione deve essere eseguita utilizzando una richiesta HTTP DELETE su /things/id, dove id è l'identificatore della TD necessariamente già esistente nella directory.

Elencazione

L'endpoint /things offre diversi modi per interrogare la raccolta di oggetti TD completi dalla directory.
Potrebbero verificarsi scenari in cui i client necessitino l'elenco in piccoli sottoinsiemi di TD. Sebbene la Search API offra la possibilità di interrogare un intervallo specifico, potrebbe non essere ottimale né adatta agli sviluppatori a questo scopo.
In modo da ottenere questo comportamento, WoTerFlow accetta tre parametri query volti a implementare un meccanismo di impaginazione dei risultati:

- **limit**: parametro per indicare il numero massimo di Thing Description da riportare nella risposta
- **offset**: parametro che indica al servizio di Discovery da dove iniziare nell'elenco dei risultati.
- **format**: questo parametro indica il formato in cui ci si aspetta la risposta. I due formati supportati sono array e collection. Mentre il primo genera semplicemente una lista di TD, il secondo è più sofisticato. Invece di fornire le TD in un array, restituisce un oggetto che racchiude i risultati, con l'aggiunta di metadati sulla ricerca stessa,i principali sono:

- **total**: numero totale di Thing Descriptions presenti nella directory
- **@id**: indica la pagina corrente. Si tratta dell'endpoint /things arricchito dei parametri offset, limit e format volti a, se utilizzati nella richiesta, ottenere la pagina corrente.
- **next**: attributo che, come il precendente, riporta l'endpoint arricchito in modo da ottenere le Thing Description contenute nella pagina successiva.
- **members**: array contenente i risultati dell'elencazione

Ad esempio, data una Directory che raggruppa un totale di 11 Thing Descriptions, se si facesse una richiesta HTTP GET a /things?format=collection&limit=10, la risposta sarà la seguente:

```
{
    "@context": "https://www.w3.org/2022/wot/td/v1.1",
    "@type": "ThingCollection",
    "total": 11,
    "@id": "/things?offset=0&limit=10&format=collection",
    "next": "/things?offset=10&limit=10&format=collection",
    "members": [
        ...
    ]
}
```

Listing 2: Elencazione in formato "collection"

Come si può notare, l'attributo "@id" determina la pagina corrente grazie al parametro offset=0 (prima TD). L'attributo "next" invece, determina la pagina successiva che conterrà l'ultima Thing Description (offset=10). Di conseguenza, per ottenere quest'ultima, sarà sufficiente fare una nuova richiesta GET al path contenuto nell'attributo "next".

Validazione

La specifica W3C non impone che vi sia una validazione prima della registrazione delle Thing Description. Tuttavia, viene fortemente consigliata, così da evitare registrazioni errate.
WoTerFlow, per ogni TD registrata o aggiornata, esegue due tipi di validazioni sulla

rappresentazione in RDF:

1. **Validazione sintattica**: validazione volta a garantire che la forma e la struttura della TD sia conforme alla relativa specifica. Ciò avvieve verificando che tutti i campi obbligatori siano presenti, che i tipi di dati siano corretti, che gli array contengono i tipi giusti di elementi e che la struttura JSON complessiva corrisponda a quanto richiesto dalla specifica.
2. **Validazione semantica**: controlla che il significato della TD abbia senso. In particolare, verifica se i concetti e le relazioni espressi nella TD siano significativi e corretti secondo ontologie e vocabolari condivisi. Ciò richiede la comprensione non solo della struttura della TD, ma anche di ciò che la TD sta cercando di descrivere di un dispositivo reale.

Per effettuare le due validazioni, WoTerFlow si serve di Shapes Constraint Language (SHACL) [16], un linguaggio per descrivere e validare grafi RDF.

3.2.2 Events

La **Notification API** serve a notificare ai client le modifiche delle TD mantenute all'interno della directory. Segue le specifiche di Server-Sent Events per fornire eventi ai client all'endpoint `/events`.
Le notifiche ai client sono solitamente composte dal tipo dell'evento, dall'id dell'evento e dall'id della TD oggetto della notifica.
La specifica W3C identifica 3 tipi di eventi:

- **thing_created**: invia un notifica ai client iscritti all'endpoint /events/thing_created ogni volta che una Thing Description viene creata nella Directory.
- **thing_updated**: notifica ai client iscritti a /events/thing_updated ogni volta che una TD è aggiornata via richiesta PUT o PATCH.
- **thing_deleted**: invia una notifica ai client iscritti a /events/thing_deleted ogni volta che una TD è eliminata.

Inoltre, se il client volesse ricevere tutte le notifiche raggruppate in un unico canale indipendentemente dal tipo di evento, lo può fare tramite una richiesta GET a /events.
Un'importante caratteristica della Notification API è la possibilità data ai client che si riconnettono di poter continuare dall'ultimo evento ricevuto. Ciò avviene fornendo l'ID dell'ultimo evento come valore dell'`Header Last-Event-ID`, grazie al quale il client riceverà tutti gli eventi persi a partire da esso.

In WoTerFlow, come discusso nella sezione 3.1.5, le funzionalità relative alla Notification API sono state implementate grazie all'utilizzo del Costrutto MutableSharedFlow. Nel dettaglio, ne vengono gestiti 3 differenti, uno per ogni tipo di canale e, ogni volta che un evento si verifica, questo viene "emesso" nel relativo flusso. Di conseguenza, tutti i client iscritti ad esso lo riceveranno.

3.2.3 Search

La **Search API** fornisce la possibilità ai client di interrogare le TD registrate nella Directory effettuando richieste all'endpoint `/search`. WoTerFlow implementa la funzionalità fornendo tre differenti modi di utilizzo: ricerca sintattica tramite XPath[17] o JsonPath[18] e ricerca semantica tramite SPARQL.

XPath

La funzionalità di ricerca sintattica tramite **XPath** è utilizzabile con richieste GET all'endpoint /search/xpath?query={xpath query}. XPath, acronimo di XML Path Language, è un linguaggio di query progettato per navigare tra elementi e attributi nei documenti XML. Il linguaggio funziona trattando i documenti come strutture ad albero, in cui ogni elemento, attributo e nodo di testo rappresenta un punto accessibile tramite una path expression, ovvero espressioni che utilizzano una sintassi simile ai percorsi del file system.

Dato che anche JSON-LD definisce una struttura ad albero, lo rende compatibile con le funzionalità offerte da XPath. Tuttavia, per essere utilizzato le Thing Description devono essere espresse in XML.
Per fare ciò, WoTerFlow si serve della libreria **FasterXML Jackson**[19] per le conversioni da JSON a XML, e della libreria **Saxon-HE**[20] per eseguire le query.
In particolare, il processo di esecuzione di una query XPath opera sulla MutableMap (cache) e, per ogni TD in essa:

1. **Conversione in XML**: la TD viene convertita in una stringa XML e successivamente in un documento XML.
2. **Esecuzione della query**: sul documento XML si esegue la query XPath.
3. **Restituzione dei risultati**: se la query del punto precedente ha prodotto risultati, si effettua un operazione di lookup nella cache per ricercare l'id della TD risultante ritornandola al client.

JSONPath

Similarmente a XPath, **JSONPath** rappresenta un linguaggio di query per estrarre informazioni da documenti JSON. Esso risulta però più naturale e appropriato per WoTerFLow rispetto a XPath, principalmente perché le TD sono definite come documenti JSON-LD. Di conseguenza, il processo di esecuzione di una query JSONPath risulta più fluido e veloce comparato a quello di XPath che necessitava di operazioni aggiuntive per la conversione in XML. Per l'implementazione di questa funzionalità ci si serve della libreria **JayWay JSONPath**[21] che permette di eseguire le query. La funzionalità di ricerca sintattica tramite JSONPath è utilizzabile con richieste GET all'endpoint /search/jsonpath?query={jsonpath query}.

SPARQL

SPARQL è un linguaggio di query progettato specificamente per il recupero e la manipolazione di dati in formato RDF. L'acronimo sta per "SPARQL Protocol and RDF Query Language" e funge da metodo standard per l'interrogazione dei dati del web semantico.
WoTerFlow per la ricerca delle TD, supporta diverse tipologie di query SPARQL, mettendo a disposizione del client modi differenti di interrogare la directory. Le tipologie supportate sono:

- **SELECT**: sono la tipologia più utilizzata. Analogamente a delle query SQL, permettono di estrarre dati dai grafi RDF che corrispondo a determinati criteri di ricerca.
- **ASK**: forniscono una risposta booleana che indica se un pattern specifico esiste all'interno del Triple Store. Queste query restituiscono solo true o false, rendendole efficienti per controlli di convalida e verifica di esistenza.
- **DESCRIBE**: recupera l'intera Thing Description associata a un determinato identificatore o risorsa. Quando un client esegue una DESCRIBE su un dispositivo specifico, vengono restituiti tutti i metadati semantici che lo descrivono.
- **CONSTRUCT**: consentono la trasformazione e la ricomposizione delle informazioni contenute nella Directory non modificandone il contenuto. In particolare, permettono di creare viste personalizzate delle TD basate su criteri specifici creando nuove rappresentazioni RDF.

Inoltre, WoTerFlow supporta i differenti tipi di query in diversi formati, i principali sono:

- **JSON**: rappresenta i risultati delle query SPARQL attraverso una struttura gerarchica composta da oggetti e array. La risposta contiene tipicamente un oggetto principale con una sezione "head" che elenca le variabili selezionate e una sezione "results" che contiene i bindings effettivi. Ogni binding corrisponde a una riga di risultati, dove le variabili sono associate ai loro valori insieme a metadati che indicano il tipo di dato.
- **XML**: utilizza una struttura gerarchica con elementi specifici che descrivono variabili, soluzioni e bindings. Ogni risultato viene incapsulato in elementi XML che specificano esplicitamente il tipo e il contenuto di ciascun valorei.
- **CSV e TSV**: Comma-Separated-Values e Tab-Separated Values rappresentano i risultati delle query SPARQL come semplici file di testo tabulari, dove ogni riga corrisponde a un risultato e i valori sono separati rispettivamente da virgole o tabulazioni. L'utilizzo di questi formati torna particolarmente utile quando si vogliono riportare i risultati su fogli di calcolo, strumenti di business intelligence o di analisi dei dati.
- **TURTLE**: rappresenta un formato di serializzazione RDF utilizzato specificamente solo per query CONSTRUCT e DESCRIBE che restituiscono grafi interi. La sintassi utilizza prefissi per abbreviare URI lunghi e permette di raggruppare triple che condividono lo stesso soggetto o soggetto-predicato, rendendo la rappresentazione più compatta, leggibile e comprensibile.

WoTerFlow espone le funzionalità di ricerca semantica tramite query SPARQL all'endpoint /search/sparql in 2 modi: o tramite POST inserendo la query nel body della richiesta, oppure tramite GET inserendo la query nel rispettivo parametro 'query'.

Capitolo 4

Implementazioni aggiuntive e refactoring

L'evoluzione di sistemi software richiede spesso interventi mirati per estendere funzionalità esistenti e risolvere problematiche specifiche. Il presente capitolo illustrerà le modifiche ed estensioni apportate a WoTerFlow.

4.1 Notifiche basate su risultati di query

Nella versione precedente di WoTerFlow, le funzionalità di notifica degli eventi, come da specifica W3C, permettevano solo di ricevere notifiche "generiche", indipendentemente dal fatto che un subscriber potesse non essere interessato a ricevere quelle di tutte le TD ma solo di alcune.

A tal proposito, si è deciso di dare la possibilità di registrare delle query SPARQL di tipo SELECT tramite una route POST aggiuntiva **events/query_notification**. Attraverso questo approccio, il client sarà notificato solo degli eventi delle TD che corrispondono ai parametri della query.

Tuttavia, per ottenere notifiche separate per ogni query, non basta più utilizzare, come nel caso degli altri tipi di eventi, un unico MutableSharedFlow. Bensì, è necessario avere un canale di notifica dedicato a ciascuna singola query. Ciò è stato possibile grazie all'utilizzo di una MutableMap di MutableSharedFlow dove le chiavi per identificare i canali sono gli id delle query e, queste ultime, sono a loro volta memorizzate in una MutableMap con l'id come chiave.

```
suspend fun addNotificationQuery(call: ApplicationCall): Long {
    var query: String? = call.receive()
    val accept = call.request.header(HttpHeaders.Accept)

    if (query.isNullOrEmpty()) {
        throw BadRequestException("The request body is empty.")
    }

    val parsedQuery = QueryFactory.create(query,
    Syntax.syntaxSPARQL_11)
    parsedQuery.addSubjectVariable()
    query = parsedQuery.toString()

    val format =
    SparqlController.validateNotificationQueryFormat(parsedQuery,
    accept)
        ?: throw UnsupportedSparqlQueryException("Mime format not
        supported")

    val queryId = queryIDsCounter.incrementAndGet()
    val notificationQuery = NotificationQuery(
        id = queryId,
        query = query,
        resultFormat = format,
    )

    queries.put(queryId, notificationQuery)
    queryNotificationSseFlow.put(queryId,MutableSharedFlow())

    return queryId
}
```

Listing 3: Metodo per la registrazione di una query di notifica

La funzione presentata nel listing 3 gestisce la logica di validazione e memorizzazione delle query di cui essere notificati, in particolare:

- **verifica** che la query non sia vuota e che sia di di tipo SELECT. Quest'ultimo controllo è necessario affinchè la query possa restituire l'id delle TD che

corrispondono ai criteri della query.

- **aggiunge** come parametro di ritorno l'id della TD nel caso non fosse stato inserito dall'utente
- **genera** un id per la query, utilizzato come chiave nelle MutableMap delle query e dei MutableSharedFlow.
- **inserisce** la query in una MutableMap, con chiave l'id appena generato.
- **crea** un nuovo MutableSharedFlow nella MutableMap dedicata, la cui chiave è l'id della query.

Per quanto riguarda la notifica degli eventi in sé, è stata implementata la seguente funzione.

```
suspend fun executeAndNotifyQueries(
    thingId: String,
    ts: ThingDescriptionService
) {
    for (id in eventController.queries.keys) {
        val query = eventController.queries[id]
        val stringQueryResult =
        ts.executeNotificationQuery(query!!)
        val jsonQueryResult = Utils.toJson(stringQueryResult)

        val bindings =
        jsonQueryResult.get("results").get("bindings")
        if (bindings.isEmpty) {
            continue
        }

        for (td in bindings) {
            val resultTdId = td.get("s").get("value").asText()

            if (resultTdId.equals(thingId)) {
                eventController.notify(
                    EventType.QUERY_NOTIFICATION,
                    "{ \n\"id\": \"${thingId}\" }",
                    id
                )
                break
            }
```

```
            }
        }
    }
```

Listing 4: Metodo per la notifica degli eventi di una particolare TD

Ogni volta che una TD viene inserita o aggiornata, si eseguono tutte le query registrate fino a quel momento. Per ciascuna di esse, si verifica che l'id ritornato corrisponda a quello della TD in questione e, se corrispondono, verrà inviata la notifica contenente l'id della TD sul canale della query.
È importante sottolineare che questa funzione non verrà mai eseguita nel caso di eliminazione di una TD. Questo perché, dal momento che la TD è eliminata, nessuna query restituirebbe un id corrispondente.

Di seguito è riportato un esempio di connessione in listening per gli eventi di tipo **query_notification**, tramite il software Postman.

Figura 4.1: SSE listening sul canale di notifica di una query tramite Postman

4.2 Modifiche degli ID delle Thing Description

Durante l'analisi e il test delle funzionalità di WoTerFlow, è emerso che un utente potesse modificare l'id di una Thing Description. Consultando la documentazione della Things API discussa nella specifica WoT Discovery, non viene fatta menzione del fatto che un client potesse o meno farlo.
Tuttavia, consentendolo, si incorrerebbe in inconsistenze rispetto alle rappresentazioni RDF memorizzate nel Triple Store dato che quest'ultima continuerà ad avere un attributo graph contenente l'id precedente, mentre il contenuto effettivo della TD avrà il nuovo id.

Ad esempio, se si inserisse la Thing description seguente tramite una PUT /things/urn:uuid:0804d572-cce8-422a-bb7c-4412fcd56f06:

```
    {
```

```
  "@context": "https://www.w3.org/2022/wot/td/v1.1",
  "@type": "Thing",
  "id": "urn:uuid:0804d572-cce8-422a-bb7c-4412fcd56f06",
  "title": "Versione originale",
  "description": "descrizione",
  "securityDefinitions": {
    "nosec_sc": {
      "scheme": "nosec"
    }
  }
}
```

Listing 5: Thing Description originale e corretta

Successivamente, se la si modificasse tramite PUT alla stessa route di prima ma inserendo nel body della richiesta:

```
{
  "@context": "https://www.w3.org/2022/wot/td/v1.1",
  "@type": "Thing",
  "id": "urn:uuid:0804d572-cce8-422a-bb7c-4412fcd56f08",
  "title": "Versione con Id cambiato",
  "description": "nell'id 8 al posto di 6 come ultima cifra",
  "securityDefinitions": {
    "nosec_sc": {
      "scheme": "nosec"
    }
  }
}
```

Listing 6: Thing Description con id modificato

allora, qual'ora si dovesse fare una GET a things/urn:uuid:0804d572-cce8-422a-bb7c-4412fcd56f08, questa non restituirà alcun risultato. Mentre se la GET la si facesse all'id originale, things/urn:uuid:0804d572-cce8-422a-bb7c-4412fcd56f06, sarà restituita come risposta la Thing description con id modificato.
Un altro importante problema legato alla modifica degli id è che, se si effettivamente inserisse una Thing Description con id 0804d572-cce8-422a-bb7c-4412fcd56f08, ciò

sarebbe possibile e dunque si avranno 2 Thing Description con id uguale.

Pertanto, considerando i numerosi problemi derivati dal consentire le modifiche degli id, e considerando che la specifica del W3C non impone vincoli implementativi in merito, è stato ritenuto ragionevole impedirlo.

4.3 Refactoring notifiche di eventi

Durante l'implementazione della funzionalità di notifica di eventi di tipo "query_notification", un'importante decisione è stata quella di come integrarla nel codice con le altre.
Infatti, le notifiche di tutti gli altri tipi di evento venivano gestite dalla classe ThingDescriptionController che, quando un inserimento, un update o un'eliminazione di una Thing Description andava a buon fine, richiamava direttamente il metodo notify() dell'EventController.

```
eventController.notify(EventType.THING_UPDATED, "{ \n\"id\":
\"${thingId}\" }")
```

Listing 7: Notifica di un evento di tipo thing_updated

Seguendo questa logica, sarebbe stato sufficiente integrare il metodo presentato in 4 nell'EventController e richiamarlo dopo il metodo di notifica degli altri eventi.
Tuttavia, al fine di favorire la manutenibilità, la testabilità e la leggibilità del codice, è stato deciso di adottare un approccio differente.
Questo approccio consiste nel creare un layer di astrazione tra il ThingDescriptionController e l'EventController così che, il primo non si debba preoccupare direttamente di quanti e quali eventi notificare e quali operazioni effettuare per farlo. Ciò è stato possibile grazie all'implementazione del design pattern strutturale Decorator[22].

Decorator permette di allegare nuovi comportamenti a degli oggetti inserendoli all'interno di altri oggetti che contengono i comportamenti desiderati. Il pattern può essere visualizzato come una struttura a strati concentrici, simile a una matrioska: ogni livello aggiunge funzionalità mantenendo l'interfaccia comune, fino al componente base.

Come raffigurato in figura 4.2, l'implementazione del pattern è costituita da:

- **EventNotifier**, l'interfaccia comune a gli oggetti decorati e a quelli decoratori. Definisce il metodo notify() che tutti gli oggetti dovranno implementare.
- **DefaultEventNotifier**, rappresenta la classe volta a gestire gli eventi "base", definiti dalla specifica W3C. A partire da essa, verranno aggiunti i nuovi comportamenti.
- **BaseEventNotifierDecorator**, ha un campo per fare riferimento a un oggetto wrapper. Il tipo del campo deve essere dichiarato come EventNotifier in modo che possa contenere sia componenti concreti che decoratori. Il decoratore di base delega tutte le operazioni all'oggetto wrapper.
- **QueryNotificationDecorator**, definisce il comportamento aggiuntivo, tramite il metodo executeAndNotifyQueries(), che può essere aggiunto dinamicamente ai componenti. Sovrascrive il metodo notify() del decoratore base ed esegue il suo comportamento dopo la chiamata al metodo padre.

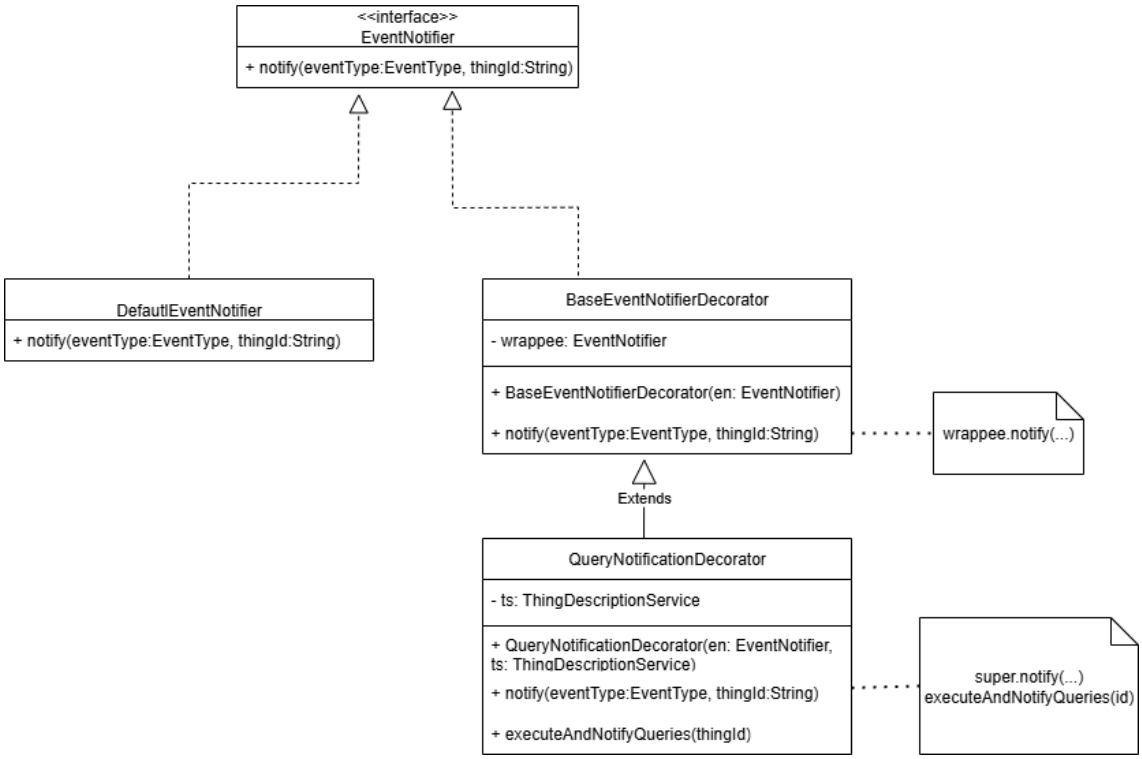

Figura 4.2: Implementazione di Decorator per la notifica degli eventi

Pertanto, con l'implementazione di decorator, nel ThingDescriptionController la gestione della notifica degli eventi sarà delegata al DefaultEventNotifier se non ci sarà necessità di eseguire, come nel caso delle eliminazioni, l'esecuzione delle query. Mentre, per tutti gli altri casi, il compito sarà delegato al QueryNotificationDecorator, responsabile di fornire il comportamento aggiuntivo:

```
val queryNotificationEventNotifier =
QueryNotificationDecorator(defaultEventNotifier, ts)

queryNotificationEventNotifier.notify(EventType.THING_UPDATED,
thingId)
```

Listing 8: Notifica di un evento thing_updated con il pattern Decorator

In conclusione, adottando questo approccio si ottiene una maggiore manutenibilità del codice dato che, qualora si decidesse di voler aggiungere un nuovo tipo di evento, basterà creare una nuova classe estendendo il decoratore base e aggiungere in essa i desiderati comportamenti aggiuntivi. Inoltre, ciò permette di rispettare a pieno il principio SOLID[23] Single Responsibility Principle, separando le responsabilità derivate dalla notifica di eventi in classi e metodi specifici.

4.4 Test aggiuntivi

Per testare il codice di WoTerFlow è stata utilizzata una test suite esterna, **Farshidtz wot-discovery-testing**[24], sviluppata in conformità alle specifiche W3C. Tuttavia, sotto alcuni aspetti, i test implementati risultano incompleti o troppo permissivi. Quindi, avendo anche implementato nuove funzionalità e aggiunto la restrizione alla modifica degli id, è stato ritenuto necessario implementare dei test aggiuntivi che si integrassero con quelli offerti dalla suite.

Per implementare i nuovi test è stata utilizzata la libreria interna di Kotlin **kotlin-test**[25], la quale fornisce annotazioni per contrassegnare le funzioni di test e un set di funzioni di utilità per eseguire asserzioni nei test, indipendentemente dal framework di test utilizzato.

4.4.1 Notifiche di eventi su query

Come ogni progetto software che si rispetti, ogni volta che si introducono nuove funzionalità è bene che esse vengano testate. Ciò risulta fondamentale al fine di evitare di rilasciare un progetto con errori e malfunzionamenti.
Pertanto, anche per WoTerFlow dopo aver aggiunto la funzionalità di eventi su query è stato necessario introdurre dei test ad-hoc.

I test implementati sono:

1. **Test di notifica base**: Verifica il meccanismo fondamentale di notifica quando una Thing Description inserita soddisfa una query SPARQL registrata. Il test stabilisce una connessione SSE, inserisce la TD tramite PUT e verifica che l'evento generato contenga l'id corretto e il tipo di evento query_notification.

2. **Test concorrenza multi-client**: Testa la capacità del sistema di gestire simultaneamente più client registrati alla stessa query. Utilizza due client HTTP separati che si registrano allo stesso endpoint SSE. Ciò verifica che la registrazione crei 2 canali differenti, e che l'inserimento di un singolo TD generi eventi distinti per entrambi i client. Così facendo si verifica che il sistema mantenga correttamente lo stato delle registrazione multiple e notifichi gli eventi in modo isolato senza conflitti.

3. **Test filtro selettivo**: Verifica che le TD che non soddisfano i criteri della query registrata non generino notifiche. Il test utilizza un TD modificato progettato per non corrispondere alla query di test. Lo scadere del timeout per la ricezione dell'evento conferma che il meccanismo di notifica funziona correttamente.

4. **Test query senza ritorno dell'id**: Testa il comportamento del sistema con query SPARQL che non includono il soggetto nella clausola SELECT. Questo scenario è importante poiché il sistema deve comunque essere in grado di identificare quale TD ha scatenato l'evento, anche quando la query non richiede esplicitamente l'URI del soggetto.

5. **Test canale eventi globale**: Valida il funzionamento dell'endpoint generico `/events` che dovrebbe ricevere tutti i tipi di eventi. Il test verifica che un singolo aggiornamento di TD generi sia l'evento `thing_updated` che `query_notification`, confermando che il canale globale riceve correttamente eventi di tutti i tipi.

6. **Test gestione errori - body vuoto**: Verifica la corretta gestione degli errori quando una richiesta non contiene una query nel body. Il sistema deve respingere la connessione con status HTTP 400 (Bad Request) e non permettere

la registrazione di query malformate che potrebbero causare comportamenti imprevisti.

7. **Test gestione errori - query non supportata**: Testa il comportamento con query SPARQL sintatticamente corrette ma non supportate dal sistema (come le query UPDATE). Il rifiuto con HTTP 400 dimostra che il sistema valida il tipo di query prima di registrare la query, prevenendo operazioni che non possono essere completate correttamente o che possano minare alla sicurezza del sistema.

8. **Test gestione errori - query invalida**: Testa la gestione di query SPARQL sintatticamente errate. Il sistema deve rilevare l'errore durante il parsing della query e restituire HTTP 500 (Internal Server Error), indicando un problema nell'elaborazione della query piuttosto che un errore di formato della richiesta.

4.4.2 Sparql

Analizzando i test implementati dalla test di suite, è stato riscontrato che non tutte le funzionalità di ricerca avanzata SPARQL e i formati di query da essa supportati fossero stati testati. Di conseguenza, è stato ritenuto necessario implementare i seguenti test:

1. **Test formato JSON**: Verifica che l'endpoint SPARQL rispetti correttamente la negoziazione del contenuto per il formato JSON. Il test invia una query SELECT con header `Accept: application/sparql-results+json` e conferma che la risposta mantenga il formato richiesto e contenga i dati attesi.

2. **Test formato XML**: Testa il supporto per il formato XML. Testa che query SELECT con `Accept: application/sparql-results+xml` producano risposte XML. Importante per compatibilità con strumenti SPARQL legacy e sistemi enterprise che privilegiano XML per l'integrazione.

3. **Test formato CSV**: Testa il supporto per il formato CSV. Questo formato è cruciale per l'analisi dei dati e l'importazione in spreadsheet o strumenti di business intelligence. Il test conferma che l'header `Accept: text/csv` generi output nel formato corretto.

4. **Test formato TSV**: Testa il supporto per il formato Tab-Separated Values, variante del CSV che utilizza tabulazioni come delimitatori. Formato spesso preferito in contesti dove i dati possono contenere virgole, garantendo parsing più affidabile in pipeline di elaborazione dati.

5. **Test fallback formato non supportato**: Verifica il comportamento quando viene richiesto un formato incompatibile con il tipo di query. Poiché Turtle

non è adatto ai risultati tabulari, una query SELECT con Accept: text/turtle deve automaticamente utilizzare il formato JSON.

6. **Test query ASK**: Testa query di tipo ASK, ovvero che restituiscono valori booleani. Il test verifica che, dopo aver inserito una TD, la risposta della query sia `true`. Mentre, quando vine eliminata, verifica che la risposta cambi da `true` a `false`.

7. **Test query DESCRIBE**: Testa query DESCRIBE, ovvero che restituiscono grafi RDF completi. Il test consiste nel verificare che, dopo l'inserimento della TD, la query restituisca un risultato. Mentre, dopo l'eliminazione, verifica che la query non ne restituisca nessuno.

8. **Test query CONSTRUCT**: Testa query CONSTRUCT, ovvero che generano nuovi grafi RDF basati su pattern specifici. Nella query di test, doveva essere generato un nuovo grafo RDF a partire da TD con uno specifico titolo. Il test conferma che, dopo l'inserimento della TD il grafo è costruito correttamente, mentre dopo la rimozione non lo è.

4.4.3 Things

L'aggiunta di ulteriori test relativi a operazioni CRUD sulle Thing Descriptions è dovuta principalmente a:

1. **Modifica degli id**: ovvero verificare che il nuovo vincolo sull'impossibilità di modificare gli id delle TD funzioni correttamente

2. **Roundtripping**: cioè la corretta traduzione di una TD da JSON-LD a RDF e viceversa. Infatti, quando inserita, la TD viene tradotta in RDF per essere memorizzata e, quando un client la recuperara dal Triple Store, essa deve essere tradotta nuovamente nella rappresentazione JSON-LD originale.

Il motivo dei test sul roundtripping risiede nel fatto che la suite di test lo riduceva ad una versione molto semplicistica e approssimativa, verificando solamente che, dopo aver inserito una TD ed averla recuperata, il titolo della prima fosse uguale a quello della seconda.

Pertanto, i principali test introdotti sono:

1. **Test gestione errori - id non corrispondente**: Sia nel caso di richieste PUT che PATCH, si verifica che il tentativo di modifica di un id non vada a buon fine e il server risponda con un errore Bad Request 400.

2. **Test roundtripping**: Testa che, dopo l'inserimento di una TD, il suo retrieval restituisca in output la stessa identica TD inserita, listing 9. Per testare

l'uguaglianza delle due, entrambe vengono trasformate in formato RDF e si verifica che siano "isomorfe". Per "isomorfe" si intende che le due rappresentazioni siano equivalenti semanticamente. Ciò implica che le TD possano avere strutture differenti, purchè il significato del contenuto sia il medesimo.

```
fun assertThingDescriptionsEquals(td1: String, td2: String) {
    val objectNode1 = jsonMapper.readValue<ObjectNode>(td1)
    val objectNode2 = jsonMapper.readValue<ObjectNode>(td2)
    objectNode2.remove("registration") // removes registration
    infos
    objectNode2.remove("@version") // removes version field

    val rdfModel1 =
    converter.toRdf(converter.toJsonLd11(objectNode1).toString())
    val rdfModel2 =
    converter.toRdf(converter.toJsonLd11(objectNode2).toString())

    if(!rdfModel1.isIsomorphicWith(rdfModel2)) {
        var json1 = jsonMapper.writeValueAsString(objectNode1)
        json1 =  JsonPrettifier.prettifyJson(json1)

        var json2 = jsonMapper.writeValueAsString(objectNode2)
        json2 =  JsonPrettifier.prettifyJson(json2)

        fail("Retrieved td does not matches the inserted
        one:\nExpected: $json1;\nGot: $json2")
    }
}
```

Listing 9: Metodo di verifica del corretto roundtripping

4.5 Roundtripping

Come accennato in precedenza, gli obiettivi principali di questo progetto erano l'estensione delle funzionalità e la risoluzione di problemi esistenti nella versione precedente di WoTerFlow.
Uno di questi problemi, se non il principale, era l'errata traduzione delle rappresentazioni RDF a quelle in JSON-LD e viceversa, nota come roundtripping.
Tale problema era noto essere dovuto ad alcuni bug presenti nella libreria Titanium

json-ld, utilizzata per effettuare le conversioni. In WoTerFlow era stato provato ad implementare un workaround volto ad aggirarli. In particolare, nella classe RDF-Converter, nel metodo toJsonLd11() mostrato nel listing 10, oltre alle conversioni dalla versione 1.0 alla 1.1 di JSON-LD, venivano effettuate sostituzioni sintattiche di campi non tradotti correttamente.

```
if (thing.has("securityDefinitions")) {
    val securityDefinitions = thing.get("securityDefinitions")
    val updatedSecurityDefinitions =
    JsonNodeFactory.instance.objectNode()

    securityDefinitions.fieldNames().forEach { fieldName ->
        val securityDefinition = securityDefinitions.get(fieldName)

        // If it is `@none`, a new node `scheme` will be set to
        `nosec_sc`.
        if (fieldName == "@none") {
            val scheme = securityDefinition.get("scheme").textValue()
            val newNode = JsonNodeFactory.instance.objectNode()
            newNode.put("scheme", scheme)
            updatedSecurityDefinitions.set("nosec_sc", newNode)
        } else {
            // Else the prefix `td:` will be removed from
            `td:scheme`.
            if (securityDefinition.has("td:scheme")) {
                val scheme = securityDefinition.get("td:scheme")
                (securityDefinition as ObjectNode).put("scheme",
                scheme.textValue())
                securityDefinition.remove("td:scheme")
            }
            updatedSecurityDefinitions.set(fieldName,
            securityDefinition)
        }
    }

    // `title` field purified from prefix.
    if (thing.has("td:title"))
        thing.replace("title", thing.remove("td:title"))

    ...
```

Listing 10: Traduzione dei campi td:title e td:scheme nel formato corretto nel metodo toJsonLd11()

Tuttavia, questo approccio non risulta sufficiente, non venendo considerati tutti i possibili campi affetti da tali bug.

L'approccio adottato in modo da risolvere i problemi precedenti consiste nel sostituire la versione della libreria Titanium json-ld contenente i bug con un'altra versione della stessa con i bug risolti. Così facendo, le conversioni delle Thing Description da RDF a JSON-LD avvengono con successo, ottenendo un roundtripping completo.

Capitolo 5

Conclusioni

Questa tesi ha esaminato la Thing Description Directory WoTerFlow, un componente fondamentale per consentire la scoperta e la gestione dei dispositivi negli ecosistemi IoT. A seguito di un'analisi completa dei concetti WoT e delle funzionalità supportate, questo lavoro ha presentato 5 modifiche all'implementazione originale: una nuova funzionalità di notifica, l'impedimento della modifica degli id delle TD, una suite di test ampliata, refactoring basato sul pattern Decorator e l'introduzione della nuova versione di Titanium JSON-LD.

In particolare, ciascuna di esse apporta specifici vantaggi a WoTerFlow, di cui:

- **Query Notification**: estende le funzionalità di notifica degli eventi sulle TD esistenti. Ciò permette di ricevere notifiche specifiche a determinati parametri, evitando di riceverne di non desiderate.
- **Modifiche id**: rende la gestione dei dati in WoTerFlow più consistente, evitando ridondanze e inconsistenze.
- **Pattern Decorator**: migliora la testabilità e modularità del codice.
- **Test aggiuntivi**: aumentano l'affidabilità e manutenibilità di WoTerFlow. I nuovi test garantiscono che le funzionalità esistenti funzionino come previsto e che le modifiche apportate non introducano bug.
- **Titanium JSON-LD**: la nuova versione introdotta consente di ottenere un roundtripping completo, rendendo WoTerFlow più affidabile e usabile.

Il presente lavoro aiuta a rendere le Thing Description Directory (TDD) più utili e manutenibili nell'ecosistema WoT. Le estensioni affrontano problemi reali di progettazione e usabilità, soddisfando l'obiettivo più ampio di consentire la scoperta e l'integrazione dei dispositivi IoT in ambienti sempre più complessi e dinamici. Man

mano che le tecnologie WoT si sviluppano e emergono nuovi requisiti, le basi stabilite da questo lavoro consentono a WoTerFlow di crescere in modo continuo.

Infine, poiché il Web of Things è caratterizzato da continui sforzi di standardizzazione che evolvono con la maturazione delle tecnologie IoT e l'emergere di nuovi requisiti, WoTerFlow è progettato per adattarsi a questi cambiamenti. Pertanto, WoTerFlow non rappresenta solo uno strumento conforme agli standard attuali, ma un contributo al processo di standardizzazione continua del Web of Things.

Bibliografia

[1] W3C - Web of Things
https://www.w3.org/WoT/

[2] W3C - WoT Discovery
https://www.w3.org/TR/wot-discovery/

[3] W3C - Thing Description
https://www.w3.org/TR/wot-thing-description11/

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

[5] W3C - Binding Templates
https://www.w3.org/TR/wot-architecture/#sec-binding-templates

[6] W3C - Servient
https://www.w3.org/TR/wot-architecture/#sec-servient-implementation

[7] W3C - RDF
https://www.w3.org/RDF/

[8] Daniele Perrella - Implementazione di una Thing Description Directory con supporto per ricerca semantica
https://amslaurea.unibo.it/id/eprint/30926/1/Tesi_Magistrale_Daniele_Perrella.pdf

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

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

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

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

[13] W3C - Sparql
https://www.w3.org/TR/sparql11-query

[14] Transazioni ACID
https://en.wikipedia.org/wiki/ACID

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

[16] Shacl
https://www.w3.org/TR/shacl/

[17] W3C - XPath
https://www.w3.org/TR/xpath-31/

[18] JsonPath
https://en.wikipedia.org/wiki/JSONPath

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

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

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

[22] Refactoring Guru - Decorator
https://refactoring.guru/design-patterns/decorator

[23] Principi SOLID
https://en.wikipedia.org/wiki/SOLID

[24] Test Suite
https://github.com/farshidtz/wot-discovery-testing

[25] Kotlin Test
https://kotlinlang.org/api/core/kotlin-test/