POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Self-Sovereign Identity per dispositivi Internet of Things

Relatore
prof. Cataldo Basile

Candidato
Antonio Santoro

Supervisore aziendale
Fondazione LINKS
Andrea Vesco, Ph.D.

Anno accademico 2020-2021

Sommario

Lo scopo di questa tesi è quello di dimostrare l'efficacia dei concetti di identità digitale applicati ai dispositivi Internet of Things. Il progetto nasce da un proof of concept realizzato dal team di Cybersecurity della Fondazione LINKS, che rappresenta un'implementazione degli standard proposti dal W3C group: Decentralized Identifiers e Verifiable Credentials.

Il framework sviluppato in questo lavoro dimostra come, seguendo il paradigma Self-Sovereign, si può definire un'ecosistema in cui i peer della rete possono comunicare in sicurezza e nel rispetto della privacy, facendo uso della propria identità digitale. In una prima fase di studio quindi, sono delineati i requisiti e i casi d'uso che pongono le basi per un'attività di progettazione. Nella trattazione viene allora illustrata la composizione delle classi che caratterizzano i nodi della rete, mediante l'utilizzo di diagrammi, mettendo in evidenza le scelte tecniche e i pattern adottati per la definizione del design orientato agli oggetti. Successivamente, viene mostrata l'implementazione del framework, e come avvengono le interazioni tra i peer della rete attraverso i canali sicuri, creati per la trasmissione confidenziale delle informazioni.

Grazie alla compilazione di una distribuzione Linux custom mediante la toolchain del progetto Yocto, il codice viene poi installato su una board di architettura i.MX6 che fa parte del banco di prova su cui avvengono i test in laboratorio. Le attività sperimentali proseguono quindi con la realizzazione di un servizio remoto, che fa uso delle funzionalità del framework ed espone API REST per servire un'applicazione dimostrativa dotata di un'interfaccia web interattiva.

Lo studio si conclude con una fase di testing e profiling volta a valutare le prestazioni del codice, prendendo in considerazione i tempi di esecuzione e la memoria utilizzata sulla board di riferimento. Sono forniti quindi i risultati, divisi per setup e caso d'uso, accompagnati da una descrizione della metodologia impiegata per effettuare le misurazioni.

In appendice a questa tesi saranno infine riportate le istruzioni per l'avvio del framework, insieme a quelle per compilare e installare l'immagine Linux realizzata, a supporto degli sviluppi futuri di questo lavoro.

Ringraziamenti

Un doveroso ringraziamento al Cybersecurity team della Fondazione LINKS per la supervisione e per aver messo a disposizione gli strumenti necessari allo svolgimento di questo lavoro. Desidero inoltre ringraziare i ricercatori Antonio Defina e Guido Gavilanes per il supporto durante le attività relative al progetto Yocto.

Indice

1 Introduzione 8
- 1.1 Sicurezza nell'Internet of Things . . . 8
- 1.2 Gestione dell'identità . . . 10
- 1.3 Il problema dell'identità digitale . . . 11
- 1.4 Scopo della tesi . . . 11

2 Identità digitale 13
- 2.1 Self-Sovereign Identity . . . 13
- 2.2 Decentralized Identifiers . . . 14
  - 2.2.1 Formato . . . 14
  - 2.2.2 DID document . . . 15
  - 2.2.3 Architettura . . . 17
- 2.3 Verifiable Credentials . . . 19
  - 2.3.1 Modello . . . 20
  - 2.3.2 Verifiable Presentation . . . 22
  - 2.3.3 Casi d'uso . . . 23
  - 2.3.4 Selective disclosure . . . 26
- 2.4 Trust over IP . . . 27
  - 2.4.1 Governance stack . . . 28

3 Design 30
- 3.1 Architettura . . . 30
- 3.2 Tipi di nodi e composizione delle classi . . . 31
  - 3.2.1 Issuer . . . 31
  - 3.2.2 Verifier e Relay . . . 34
  - 3.2.3 Holder . . . 36
  - 3.2.4 Tangle . . . 36
- 3.3 Funzionalità di rete . . . 38
  - 3.3.1 Descrizione del modulo . . . 38
  - 3.3.2 Modelli e mapping . . . 40

4 Implementazione 42
- 4.1 Aspetti implementativi . . . 42
  - 4.1.1 Gestione della persistenza . . . 42

4.1.2 Gestione delle richieste di rete . . . 48
4.1.3 Concorrenza delle richieste . . . 54
4.2 Casi d'uso . . . 55
4.2.1 Inizializzazione . . . 55
4.2.2 Autenticazione e canali sicuri . . . 58
4.2.3 Gestione della status list . . . 61
4.2.4 Rilascio di credenziale . . . 65
4.2.5 Accesso al servizio . . . 67
4.2.6 Revoca di credenziale . . . 69
4.2.7 Inoltro di richiesta . . . 70

5 Sperimentazione 73
5.1 Yocto . . . 73
5.1.1 Componenti e strumenti di sviluppo . . . 74
5.1.2 Workflow . . . 77
5.2 Testbed e setup . . . 79
5.3 Implementazione del servizio . . . 81
5.3.1 Ricezione delle richieste e gestione delle eccezioni . . . 84
5.3.2 API . . . 86
5.3.3 Esempio di richiesta al servizio . . . 89
5.4 Applicazione dimostrativa . . . 91
5.4.1 Angular e struttura dell'applicazione . . . 92
5.4.2 Navigazione . . . 93
5.4.3 Presentazione dei contenuti . . . 99
5.4.4 Richieste di rete e flusso di dati . . . 110
5.4.5 Aggiornamento dei contenuti . . . 116

6 Risultati 124
6.1 Testing del framework . . . 124
6.1.1 Funzioni mock . . . 125
6.1.2 Classi mock . . . 127
6.1.3 Esempio . . . 129
6.2 Profiling . . . 130
6.2.1 Profiling della memoria . . . 131
6.2.2 Utilizzo memoria . . . 134
6.2.3 Profiling CPU . . . 135
6.2.4 Tempi di esecuzione . . . 137

A Setup, build e installazione 143
A.1 Compilazione con Yocto . . . 143
A.1.1 Installazione dei requisiti . . . 143
A.1.2 Download di Yocto Dunfell . . . 143
A.1.3 Inizializzazione directory di build . . . 144
A.1.4 Integrazione di livelli aggiuntivi . . . 145
A.1.5 Compilazione . . . 147

A.2 Installazione dell'immagine 147
A.2.1 Preparazione della SD card 147

B Istruzioni per l'esecuzione 150
B.1 Avvio del framework 150
B.1.1 Tangle 150
B.1.2 Issuer 151
B.1.3 Verifier 151
B.1.4 Relay 152
B.1.5 Holder 153
B.1.6 Servizio remoto 153
B.2 Test suite 154
B.3 Esecuzione del dimostrativo 154
B.3.1 Installazione dei package 154
B.3.2 Avvio server locale con Webpack 155
B.3.3 Deploy su server nginx 155

Bibliografia 157

Elenco delle figure 163

Elenco delle tabelle 165

Capitolo 1

Introduzione

Internet of Things (IoT) è un termine che indica l'interconnessione di entità eterogenee, dove il termine entità si riferisce a esseri umani, sensori, o qualunque cosa possa richiedere o fornire un servizio [9]. Sebbene non ci siano cifre consistenti sul numero di dispositivi IoT connessi, si può comunque affermare che i numeri tendano a crescere rapidamente [21]. L'avanzamento tecnologico è stato tale da portare i dispositivi IoT ad essere applicati in numerosi ambiti, spesso però implementando soluzioni e standard proprietari. La mancanza di una visione comune, combinata con l'evoluzione dello sviluppo, ha dato vita ad un incremento dei fattori di rischio che rappresentano una minaccia per la sicurezza dei dispositivi e per chi ne fa uso.

In ogni area di applicazione, i dispositivi IoT vengono tipicamente dotati di risorse limitate, che vanno a definire un vincolo sulla velocità di elaborazione delle informazioni. Siccome i dati vengono protetti facendo uso di firme digitali, una public key infrastructure è fondamentale ma introduce un'ulteriore sfida, dato che le operazioni con le chiavi a crittografia pubblica richiedono risorse computazionali e di memoria che alcuni dispositivi non offrono.

Un altro fattore da considerare è l'eterogeneità delle specifiche hardware che caratterizzano i dispositivi IoT, la sicurezza deve infatti essere implementata mediante soluzioni compatibili con stack tecnologici diversi. Purtroppo però, è pratica comune adattare queste soluzioni a delle "scatole chiuse", secondo un principio sbagliato che fa della segretezza (security through obscurity) il suo meccanismo di protezione.

1.1 Sicurezza nell'Internet of Things

I dispositivi Internet of Things gestiscono un'enorme mole di dati e offrono servizi in ambiti che si estendono dall'industria manifatturiera fino ai sistemi sanitari. Tuttavia, alla luce delle numerose evidenze scientifiche, questi dispositivi introducono dei rischi che possono potenzialmente rappresentare una minaccia, sia per l'ambiente in cui sono adoperati, sia per la privacy di chi li utilizza, dato che spesso processano informazioni sensibili. È dunque fondamentale che i sistemi in cui è forte la presenza di questi dispositivi sia sicuro.

Requisito	**Definizione**
Confidenzialità (Confidentiality)	Garantisce che solo gli utenti autorizzati abbiano accesso alle informazioni
Integrità (Integrity)	Garantisce che non ci siano manipolazioni sui dati
Disponibilità (Availability)	Garantisce che tutti i sistemi siano disponibili quando richiesti dagli utenti autorizzati
Responsabilità (Accountability)	Abilità di un sistema a stabilire la responsabilità delle azioni da parte degli utenti
Verificabilità (Auditability)	Abilità di un sistema a monitorare tutte le azioni
Affidabilità (Trustworthiness)	Abilità di un sistema a verificare la fiducia verso un'entità terza
Non ripudio (Non-repudiation)	Abilità di un sistema a confermare l'occorrenza, o la non occorrenza, di un'azione
Privacy	Garantisce il rispetto della privacy consentendo agli individui il controllo delle informazioni personali

Tabella 1.1: Requisiti di sicurezza.

Tipicamente, i requisiti di sicurezza si suddividono in tre macro-categorie: confidenzialità, integrità e disponibilità (triade CIA). La confidenzialità definisce un set di regole che limitano gli accessi non autorizzati a certe informazioni. L'integrità è necessaria per garantire l'affidabilità dei servizi erogati, c'è bisogno infatti che le informazioni collezionate e processate siano legittime. La disponibilità dei dispositivi è fondamentale per garantire che il sistema sia pienamente funzionante, e che i servizi siano erogati senza interruzioni. In Tabella 1.1 viene proposta un'estensione della triade CIA [22] che riporta i requisiti tali per cui un sistema può essere considerato *sicuro*.

La ricerca di soluzioni, per realizzare infrastrutture che garantiscono i requisiti di sicurezza menzionati, ovviamente non è esente da sfide. I temi da affrontare [49] possono essere elencati come segue:

- Eterogeneità dei dispositivi. I protocolli di sicurezza devono essere compatibili con le diverse tipologie di dispositivo e con i relativi stack tecnologici, sia per quelli acceduti

in maniera diretta che per quelli che fanno da tramite.

- Risorse limitate. È essenziale implementare algoritmi crittografici efficienti che garantiscano un alto throughput e dei protocolli di sicurezza leggeri.
- Gestione dell'identità. In questo contesto rientrano tutti i problemi relativi all'autenticazione dei dispositivi, alla distribuzione delle chiavi a crittografia pubblica, e al controllo degli accessi.
- Rispetto della privacy. Le entità devono avere la facoltà di rimanere anonime in un mondo interconnesso, ma soprattutto di essere consapevoli dell'uso che viene fatto dei dati raccolti. È fondamentale quindi che vengano seguiti i principi della privacy by design.
- Fiducia e governo. La fiducia si riferisce a quella esistente nelle interazioni tra le entità, e quella che gli utenti pongono nell'utilizzo che fanno dei dispositivi. La sfera governativa invece, fa da garante per la stabilità, ed emana le norme di sicurezza che i sistemi devono rispettare.
- Tolleranza ai guasti. La sicurezza dovrebbe essere implementata by default, non solo per garantire dei meccanismi di prevenzione verso le intrusioni, ma anche per proteggere i dispositivi dagli attacchi, attraverso l'implementazione di soluzioni robuste.

1.2 Gestione dell'identità

L'autenticazione è una proprietà cruciale per i sistemi, senza di essa le altre proprietà di confidenzialità, integrità e disponibilità sono compromesse. Se infatti un attaccante riesce ad autenticarsi correttamente in un sistema, può avere accesso a qualsiasi informazione (confidenzialità), modificarla (integrità) o eliminarla (disponibilità). Ad oggi, la forma di autenticazione più comune è rappresentata da una combinazione di username e password, che non è adatta considerando l'ubiquità dei dispositivi IoT. La gestione dell'identità presenta quindi numerose sfide, dovute anche al controllo degli accessi e delle autorizzazioni per i servizi offerti. Questo problema assume una certa rilevanza nel momento in cui i sistemi sono perennemente connessi ad Internet, il che li rende potenzialmente accessibili da chiunque. Infatti più servizi vengono offerti e più è grande la superficie di attacco. Per far fronte ai problemi di autorizzazione allora occorre implementare misure di autenticazione *forti*, mediante una gestione dei ruoli granulare, capace di consentire l'accesso solo alle entità autenticate e fidate. Senza quindi dei processi di autenticazione efficaci, non è possibile essere certi di stare ricevendo dati originati dal dispositivo atteso, e lo stesso vale per i servizi offerti, poiché possono avere accesso ai dati trasmessi.

Oltre alle modalità con cui l'identità viene utilizzata per accedere ai servizi, bisogna considerare anche come questa debba essere costruita. Oggi l'identità digitale si basa su un modello centralizzato, in cui un'autorità (centrale) regola le interazioni tra i dispositivi, che possono avvenire con quelli appartenenti alle altre autorità. La maggioranza di queste interazioni avviene secondo modalità peer-to-peer, in cui la presenza della central authority

potrebbe essere eliminata. Difatti, l'introduzione di un modello di identità digitale costruito su dati accessibili in maniera decentralizzata, fa sì che ogni dispositivo possa gestire in maniera autonoma il controllo degli accessi ai servizi offerti, della propria identità digitale e dei dati prodotti.

Le Distributed Ledger Technology (come Blockchain o Tangle) permettono di realizzare reti decentralizzate basate su un registro verificabile distribuito, a cui tutti i nodi della rete possono accedere. Tecnologie simili sono quindi perfette per realizzare le fondamenta di un infrastruttura decentralizzata, in cui ogni dispositivo è riconoscibile in maniera univoca mediante l'utilizzo di identificativi crittografici. Grazie infatti all'immutabilità del registro, i peer possono fare riferimento ad una fonte di verità verso cui pongono la propria fiducia. La combinazione di queste proprietà da quindi vita a nuovi protocolli, tali da garantire comunicazioni sicure, in cui la presenza di un'autorità centralizzata non è più necessaria.

1.3 Il problema dell'identità digitale

Internet è stato progettato senza uno standard che definisse il modo per identificare persone, cose, oppure organizzazioni. Ogni piattaforma presente in rete offre quindi il proprio sistema di autenticazione, basato su una combinazione di username e password: la soluzione predominante fino ad oggi. La continua espansione della rete e l'aumento dei servizi offerti su Internet ha costretto gli utenti a registrare per ogni sito un'identità diversa, il che ha dato vita a gigantesche honeypot di dati. Ciò, oltre a rappresentare un pericolo per la sicurezza e la privacy degli utenti, non concede il controllo di questi dati a chi veramente appartengono. Gli utenti non hanno un'identità digitale unica e consolidata, bensì frammenti gestiti per ogni organizzazione.

Ad oggi non esiste un'infrastruttura *trusted*, sia dal punto di vista crittografico che governativo, che permetta al singolo di decidere come condividere la propria identità. Questa infatti dovrebbe essere costituita da un insieme di attributi verificabili in maniera globale, da divulgare solo tramite il consenso di chi la detiene. Affinché ciò sia possibile allora, occorre seguire un approccio all'identità *self-sovereign*. Con la diffusione delle Distributed Ledger Technology allora, le organizzazioni possono interagire tramite una rete distribuita, in cui le informazioni sono replicate in più luoghi, siano resistenti alla manipolazione e agli errori, e in cui i singoli peer della rete abbiano la libertà di scegliere quali dati condividere e con chi.

1.4 Scopo della tesi

Lo scopo della tesi è quello di applicare i concetti di identità digitale ai dispositivi Internet of Things, attraverso l'implementazione di un framework che realizza un'ecosistema in cui i peer possano interagire in maniera sicura facendo uso della propria identità digitale.

Il lavoro introduce i modelli, gli standard e i casi d'uso sviluppati per la gestione dell'identità dei dispositivi nel Capitolo 2. Il Capitolo 3 descrive il design realizzato secondo un approccio orientato agli oggetti, che rende il framework estensibile e conforme alla flessibilità degli standard adottati. I casi d'uso sono trattati prendendo in esame le relazioni che intercorrono tra le classi del sistema, mediante l'utilizzo di diagrammi che

mettono in luce la composizione dei nodi dell'ecosistema. Nel Capitolo 4 la trattazione prosegue descrivendo l'implementazione tramite frammenti di codice, motivando le scelte tecniche fatte e introducendo le librerie e i tool utilizzati. Il Capitolo 5 presenta il testbed su cui sono avvenute le attività sperimentali di questo lavoro. All'inizio viene delineato il workflow per la configurazione di una board di architettura i.MX6, su cui è stata installata una distribuzione Linux custom compilata mediante la toolchain del progetto Yocto. In seguito, viene presentato lo sviluppo di un servizio adatto a dispositivi con risorse limitate, che permette di gestire l'identità digitale da remoto esponendo API REST, e che serve un'applicazione internet impiegata per fini dimostrativi. L'elaborato si conclude quindi nel Capitolo 6 in cui vengono esposti i risultati ottenuti e le metodologie adottate. Infine, in appendice vengono riportate le istruzioni per riprodurre il setup utilizzato e per avviare il framework sviluppato in questa tesi.

Capitolo 2

Identità digitale

In questo capitolo verranno introdotti gli argomenti alla base del progetto di tesi. Verrà fatta una panoramica dei modelli proposti dagli standard implementati, dell'architettura generale e dei nodi che interagiscono nell'ecosistema. Sarà inoltre affrontato il paradigma Self-Sovereign che introduce ad un nuovo modello di identità digitale, illustrando le componenti che ne abilitano l'utilizzo, dal punto di vista tecnico e governativo.

2.1 Self-Sovereign Identity

Il termine Self-Sovereign Identity [45] definisce un approccio tale per cui persone, organizzazioni o cose, hanno il pieno controllo della propria identità digitale. Tale paradigma introduce ad un modello in cui il singolo è sovrano della propria identità, ed è libero dalle dipendenze verso un'autorità centralizzata.

Ad oggi, per un individuo, dover creare la propria identità digitale significa rivolgersi ad un'organizzazione e sottostare ai suoi termini di utilizzo. Questo significa che il ciclo di vita dell'identità creata è direttamente legata a quella dell'organizzazione: nel momento in cui questa cessa di esistere, con essa anche la nostra identità digitale va incontro alla stessa fine. Oltre a questo, l'organizzazione può avvalersi della facoltà di eliminarla, provocando quindi la cancellazione dell'identità della persona in rete, dato che è impossibile da rimpiazzare e da recuperare [61].

Per far fronte all'enorme mole di dati accumulata nei database delle organizzazioni, sono nati i cosiddetti identity provider. Sono onnipresenti sulla rete provider come Facebook, o Google, che offrono agli utenti un servizio di accesso, basata su un modello di identità digitale che introduce una terza entità, posta tra l'utente e l'organizzazione che eroga il servizio, che in realtà non è necessaria, se tra due entità in comunicazione vige una fiducia reciproca. Purtroppo però, in rete i protocolli di mutua autenticazione sono basati su un'infrastruttura di chiavi a crittografia pubblica che necessita di un'autorità centralizzata (CA). Gli utenti devono quindi rivolgersi alla CA per ottenere un certificato digitale, necessario per provare l'autenticità dei messaggi scambiati mediante le firme digitali. Grazie però alla diffusione delle Distributed Ledger Technology (DLT), questo non è più necessario, infatti le istituzioni e le organizzazioni governative possono lavorare insieme per

formare una rete decentralizzata, in cui i dati sono replicati e protetti da manomissioni [61]. Questa nuova infrastruttura si può allora definire Distributed Public Key Infrastructure (DKPI), grazie alla quale un'entità sulla rete è in grado di creare la propria identità in maniera autonoma e di essere libera di determinarne il ciclo di vita [44].

Gli elementi fondamentali che rendono possibile il paradigma della Self-Sovereign Identity (SSI) sono quindi:

- DID: identificativi decentralizzati;
- DKMS: un sistema di key management decentralizzato;
- DIDComm: un protocollo di autenticazione tra due peer;
- Verifiable Credentials: insieme di attributi di un soggetto verificabili in modo crittografico.

Un'ecosistema composto dagli elementi appena descritti permette alle entità che ne fanno parte di interagire creando canali sicuri, attraverso protocolli di mutua autenticazione. Le informazioni necessarie per scambiare in maniera sicura i messaggi possono essere recuperate da un registro verificabile, utilizzato per replicare le risorse a provare l'identità dei peer che popolano l'ecosistema. Ogni peer è in grado quindi di creare la propria identità digitale in maniera indipendente senza coinvolgere alcuna autorità centrale.

2.2 Decentralized Identifiers

Un DID [47] è un identificativo che permette ad un'entità di poter essere riconosciuta in maniera digitale, decentralizzata e univoca. L'identificativo viene generato in maniera autonoma da chi ne possiede il controllo, ed è stato pensato per poter essere separato da registri centralizzati, Identity Provider e Certificate Authority. Il controllo di un DID da parte di un'entità è dimostrabile in maniera crittografica senza dover interagire con un'organizzazione terza, ma solo con l'entità con cui si vuole comunicare.

I DID si presentano nella forma di una stringa, un URI, che viene utilizzata per identificare i DID document: risorse contenenti il materiale crittografico che permette ad un'entità di provare la sua autenticità e la sua associazione con il DID.

Lo standard non impone vincoli sul numero di DID posseduti. L'entità decide quali utilizzare per ogni connessione, assumendosi la responsabilità sul rischio di poter essere riconosciuto dagli altri peer. La novità è che lo standard propone un modello pensato per rispettare la privacy delle parti comunicanti *by design* [47], secondo un approccio preventivo.

I peer possono ottenere i document associati ai DID tramite l'operazione di resolution eseguita sul registro verificabile (Verifiable Data Registry), su cui sono salvate tutte le informazioni necessarie ad aprire connessioni sicure peer-to-peer.

2.2.1 Formato

Un DID è composto di tre parti:

1. Scheme
2. DID Method
3. DID Method specific-identifier

Figura 2.1: Esempio di decentralized identifier (DID). Immagine tratta da [47].

Lo Scheme è un URI che denota la sintassi dell'identificativo e il suo schema. Il DID Method specifica l'infrastruttura su cui avvengono le operazioni di associazione tra DID e DID document. Infine, il DID Method specific-identifier rappresenta un identificativo univoco all'interno del DID Method.

Il DID Method indica l'implementazione concreta del sistema che include un registro verificabile, come ad esempio tecnologie basate su blockchain [1] (Bitcoin, Ethereum...) o su tangle (IOTA). Il DID Method oltre a fissare la forma degli identificativi, determina gli elementi contenuti all'interno dei DID document e le operazioni CRUD per gestirne la persistenza.

2.2.2 DID document

L'operazione di recupero del DID document a partire dal DID è chiamata resolve. Oltre al documento, la resolve fornisce come output informazioni aggiuntive relative al document stesso, come scadenza del materiale crittografico in esso contenuto, tipo di contenuto, oppure eventuali errori scaturiti dall'operazione. Queste informazioni vengono incluse in un oggetto definito Metadata.

All'interno del DID document, il materiale crittografico è incluso in appositi campi, il cui nome indica l'utilizzo che ne viene fatto. Gli attributi fondamentali che sono stati utilizzati per il progetto sono:

- `authentication`
- `assertionMethod`
- `verificationMethod`

Come vediamo nel Listato 2.1, ogni proprietà contiene un id univoco, un campo che ne indica il tipo, e dati aggiuntivi, relativi alla semantica del campo che li contiene. La proprietà `authentication` permette di specificare chiavi a crittografia pubblica, come ad esempio chiavi RSA, la cui componente pubblica è inserita all'interno di `publicKeyPem`. Ogni chiave è identificata da una stringa formata dal DID del controller e un fragment ("...#keys-1"), che la rende univoca all'interno del documento, mentre invece la sua tipologia è riconoscibile dall'attributo `type`.

La semantica dei campi contenenti il materiale crittografico, viene definita Verification Relationship, che descrive la relazione che intercorre tra il subject del document, cioè l'entità identificata dal DID, e il metodo di verifica. `authentication` indica che il materiale contenuto al suo interno deve essere utilizzato per i processi di autenticazione: durante una comunicazione, un peer può provare di detenere il controllo del DID, mediante l'utilizzo della chiave privata reciproca di quella pubblica contenuta nel DID document. `assertionMethod` invece, può contenere una chiave che permette ad un peer di verificare i dati firmati dal controller del document. Infine, la proprietà `verificationMethod` è opzionale, in essa può essere inserito diverso materiale crittografico da riutilizzare per scopi multipli. Ad esempio, se una chiave crittografica si trova all'interno del campo authentication, può essere utilizzata *solo* per quel tipo di operazione e non per la verifica di firme digitali. Inserendo invece una chiave all'interno di un `verificationMethod`, è possibile utilizzarla per più scopi.

Chi detiene il controllo di un DID document ne potrebbe essere anche il subject, cioè l'entità descritta nel document attraverso il campo `id`. Tuttavia, un DID document può contenere un campo aggiuntivo definito `controller`, che può riferirsi anche ad un'altra entità che non viene descritta nel document. Per questo motivo, possiamo definire il controller come colui che ha la facoltà di poter aggiornare o cancellare il document. Tale proprietà è molto utile, poiché consente all'entità in possesso di un DID document di agire per conto di un'altra entità.

Il campo `services` esprime le modalità con cui stabilire una comunicazione con il subject del DID o le entità associate. Un servizio è composto da un `type` che ne descrive la semantica, un indirizzo raggiungibile in `serviceEndpoint` e un identificativo unico `id`.

Il `@context` identifica una risorsa che descrive lo schema del DID document. `created` invece include un timestamp che indica la creazione del DID document sul distributed ledger.

Per finire, lo standard prevede ulteriori campi che non saranno approfonditi in questa trattazione. Ne citiamo alcuni per dare un'idea delle possibilità date ai controller, come ad esempio delegare l'utilizzo del document, specificare controller multipli, oppure definire del materiale crittografico per protocolli di key agreement.

```
1  {
2      "@context": [
3          "https://www.w3.org/ns/did/v1"
4      ],
5      "id": "did:ott:POLYARUCUR9ANFHKACMBZIBONMMTDXBRYG9CCGRZTOE9QLDOQKFVTNAOWSP
       ↪  XGIPPQDNQDPFWVT9KSERQN",
6      "created": "2021-10-25T18:43:48.074918",
7      "authentication": {
8          "id": "did:ott:POLYARUCUR9ANFHKACMBZIBONMMTDXBRYG9CCGRZTOE9QLDOQKFVTNA
           ↪  OWSPXGIPPQDNQDPFWVT9KSERQN#keys-1",
9          "type": "RsaVerificationKey2018",
10         "controller": "did:ott:POLYARUCUR9ANFHKACMBZIBONMMTDXBRYG9CCGRZTOE9QLD
           ↪  OQKFVTNAOWSPXGIPPQDNQDPFWVT9KSERQN",
```

```
11          "publicKeyPem": "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFA
            ↪ AOCAQ8AMIIBCgKCAQEA0n12lUROi9VVZoFDan1O...\n-----END PUBLIC
            ↪ KEY-----\n"
12      },
13      "assertionMethod": {
14          "id": "did:ott:POLYARUCUR9ANFHKACMBZIBONMMTDXBRYG9CCGRZTOE9QLDOQKFVTNA
            ↪ OWSPXGIPPQDNQDPFWVT9KSERQN#keys-2",
15          "type": "RsaVerificationKey2018",
16          "controller": "did:ott:POLYARUCUR9ANFHKACMBZIBONMMTDXBRYG9CCGRZTOE9QLD
            ↪ OQKFVTNAOWSPXGIPPQDNQDPFWVT9KSERQN",
17          "publicKeyPem": "-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFA
            ↪ AOCAQ8AMIIBCgKCAQEAzYhPXrHwvGE9bq8hTzA1...\n-----END PUBLIC
            ↪ KEY-----\n"
18      },
19      "services": [
20          {
21              "id": "did:ott:POLYARUCUR9ANFHKACMBZIBONMMTDXBRYG9CCGRZTOE9QLDOQKF
                ↪ VTNAOWSPXGIPPQDNQDPFWVT9KSERQN#use-device",
22              "type": "UseDevice",
23              "serviceEndpoint": "https://example.it/use-device"
24          }
25      ]
26  }
```

Listato 2.1: Esempio di DID document in formato JSON.

2.2.3 Architettura

Nell'architettura dei Decentralized Identifiers, il DID Resolver è la componente che consente di leggere tramite la resolve un DID dal Verifiable Data Registry. L'operazione accetta in input un DID e produce in output un DID document in un formato canonico. La forma che descrive un DID document è detta rappresentazione. Questa è ottenuta tramite il processo di production, che converte i dati contenuti nel document in formati serializzabili. Mentre invece la consumption realizza l'operazione inversa, e trasforma la rappresentazione dei dati in un modello canonico. Le due operazioni permettono quindi di convertire i DID document da una forma all'altra, serializzando i dati in formati come JSON, YAML o XML.

Le risorse inserite nei DID document possono essere filtrate grazie ad alcuni parametri. Ogni risorsa è identificabile univocamente tramite un DID URL che indica la sua locazione in rete. Il DID URL è composto da:

- Did.
- Path.
- Query.

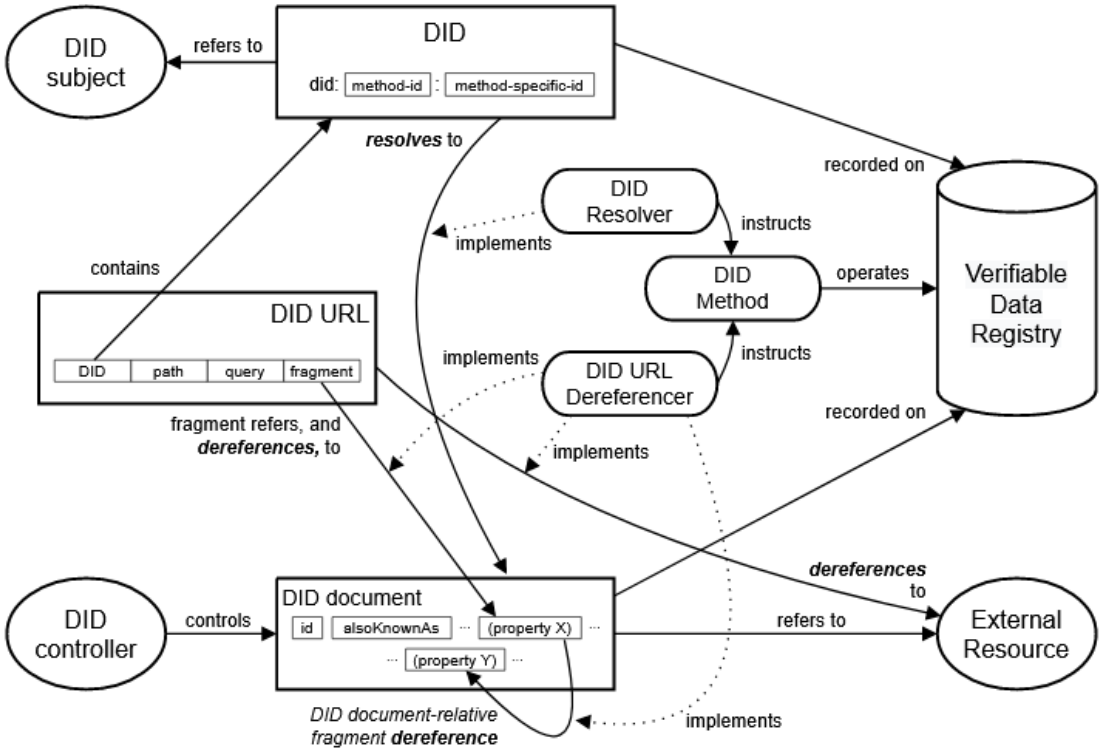


Figura 2.2: Architettura dettagliata dei Decentralized Identifier e relazioni tra le componenti. Immagine tratta da [47].

- Fragment.

Solo il DID è obbligatorio, gli altri parametri sono opzionali ma estendono l'URL, per consentire a chi esegue la resolve di applicare una selezione tra le risorse contenute nel document associato al DID.

Il path viene inserito dopo il carattere / e viene utilizzato per specializzare la semantica delle operazioni offerte dai DID Method. A seguire, le query sono specificate utilizzando il carattere di punto interrogativo ?, e vengono composte dai parametri di selezione, ognuno separato con un simbolo &. Infine, il fragment rende univoco l'identificativo della risorsa a cui si riferisce ed è rappresentato da una stringa dopo il carattere # alla fine dell'URL.

```
1 did:example:123456/path?versionId=1&?versionTime=2021-05-10T17:00:00Z#keys-2
```

Listato 2.2: Esempio di DID URL.

Il DID URL (Listato 2.2) permette quindi, mediante un Resolver, di poter ottenere una risorsa contenuta nel document facendo una dereference dell'URL.

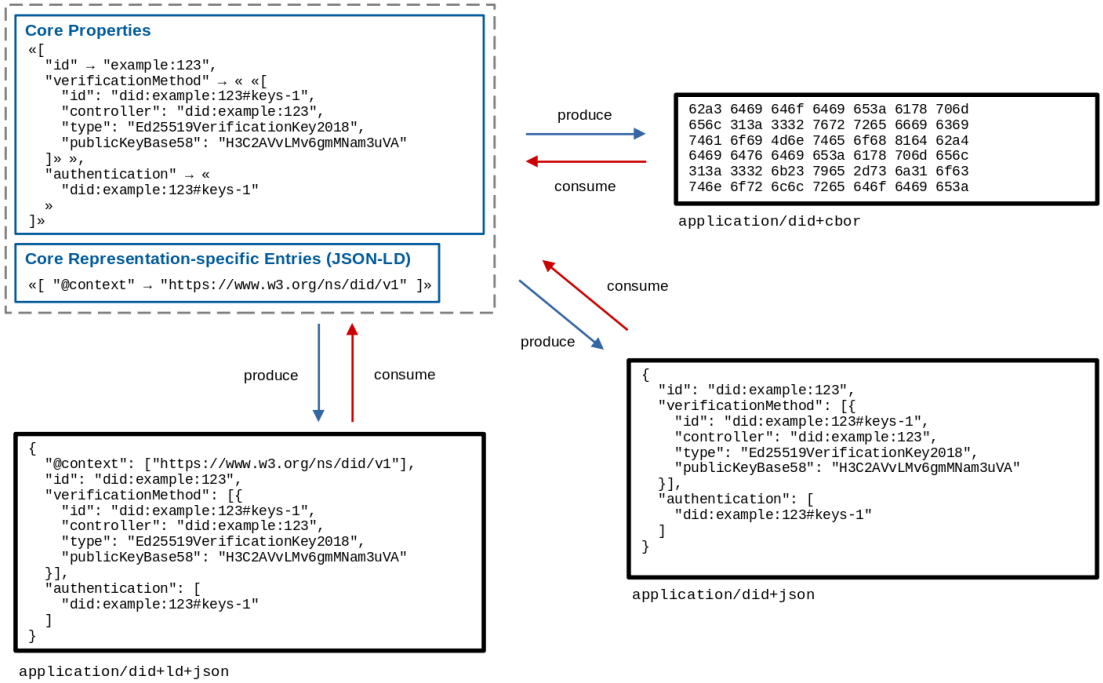


Figura 2.3: Production e consumption di rappresentazioni diverse. Immagine tratta da [47].

2.3 Verifiable Credentials

Una Verifiable Credential [46] è una struttura dati contenente un insieme di attributi relativi ad un'entità, serializzati in un formato verificabile in maniera crittografica, e che garantisce l'interoperabilità tra i sistemi.

A differenza delle credenziali fisiche, un modello di credenziale digitale gode dei seguenti vantaggi:

- Non sono facili da falsificare o da forgiare.
- Non possono essere perse o andate distrutte.
- Non sono costose da rilasciare.
- Sono scalabili.
- Sono facili da verificare online.
- Non viene mostrato più di ciò che è dovuto.
- L'integrità dei dati è protetta.

Nell'ecosistema Self-Sovereign, interagiscono diversi tipi di nodo, ognuno con un ruolo differente:

- Issuer: entità che crea e rilascia credenziali.
- Holder: riceve, mantiene e condivide le sue credenziali.
- Verifier (o Relying Party): offre servizi agli Holder accessibili tramite le credenziali, e verifica in maniera crittografica che le credenziali presentate siano autentiche e integre.

Non c'è un vincolo tale per cui un peer debba assumere un unico ruolo. Un nodo può accedere ai servizi offerti dai Verifier come Holder, e contemporaneamente offrire servizi da Verifier; un nodo può essere impiegato per rilasciare credenziali, ma può avere la necessità di accedere ad un Verifier come Holder.

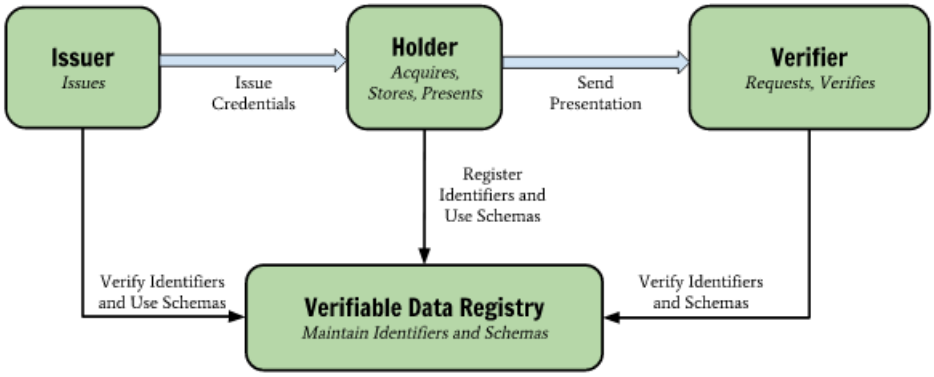


Figura 2.4: Attori nell'ecosistema delle Verifiable Credential. Immagine tratta da [46].

2.3.1 Modello

Il modello delle Verifiable Credentials è pensato per contenere un numero arbitrario di campi, il cui insieme descrive un'informazione che caratterizza l'entità a cui si riferisce. Il formato, oltre a contenere affermazioni che riguardano il soggetto della credenziale, include una proof che protegge l'autenticità e l'integrità dei dati contenuti.

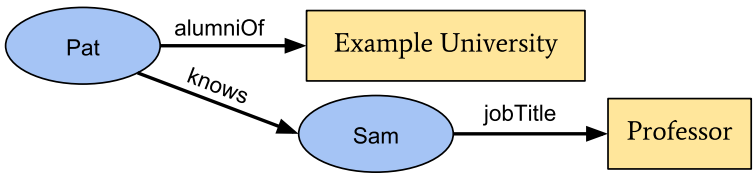


Figura 2.5: Claim multipli che esprimono un'informazione contenuta in una Verifiable Credential. Immagine tratta da [46].

Consideriamo l'esempio nel Listato 2.3, che mostra un tipo di credenziale utilizzato nel progetto. Quando una Verifiable Credential viene trasmessa, è necessario che le informazioni contenute al suo interno siano ricostruibili in una forma attesa. Siccome il modello può contenere un numero variabile di campi, oppure implementarne di nuovi non standard, occorre una risorsa che ne descriva lo schema. Il campo `@context` permette quindi di indicare una serie di URI che identificano in rete gli schemi della Verifiable Credential trasmessa.

La proprietà `type` specifica i tipi che caratterizzano la credenziale.

Chi rilascia la credenziale viene indicato nel campo `issuer` con un identificativo univoco. I Verifier fanno uso di questa informazione quando devono verificare chi ha rilasciato la credenziale. Questo significa che solo le credenziali firmate dagli Issuer di cui si fidano possono essere accettate per accedere.

Lo stato di una credenziale è verificabile facendo uso delle informazioni contenute nel `credentialStatus`. Le proprietà sono arbitrarie e dipendono dall'Issuer che le rilascia. Quelle mostrate nell'esempio verranno approfondite nel Capitolo 4, insieme ad una spiegazione dettagliata dell'implementazione.

`issuanceDate` ed `expirationDate` contengono i timestamp che indicano, rispettivamente, il momento in cui la credenziale è stata rilasciata e la relativa scadenza.

`credentialSchema` definisce lo schema con cui i Verifier verificano la struttura e i contenuti inclusi nella credenziale. Permette inoltre di specificare schemi alternativi di codifica con cui i dati vengono trasmessi, abilitando ulteriori meccanismi di verifica in zero-knowledge.

La `proof` è una struttura che viene creata dall'Issuer al momento del rilascio, e contiene: una signature JWS, lo scopo della signature (`proofPurpose`) e la chiave da utilizzare per verificare la signature (`verificationMethod`).

Il `credentialSubject` consente di esprimere un'affermazione (claim) riguardo un subject. Al suo interno sono inserite stringhe che descrivono una relazione tra il soggetto e un valore, legati da una proprietà. Questi legami formano allora delle informazioni e possono essere collegati ad altri claim. Il nostro esempio esprime un'affermazione che descrive il soggetto della credenziale come un dispositivo appartenente alla categoria dei sensori (`category`), identificato da un did (`id`), e in possesso di un certo tipo di Verifiable Credential (`type`) che gli permette di eseguire una determinata azione (`action`). In altre parole, le informazioni contenute nel `credentialSubject`, combinate al `type`, definiscono un tipo personalizzato di credenziale che una determinata applicazione accetta per l'accesso ai servizi erogati.

In conclusione, possiamo definire una Verifiable Credential come un documento contenente un set di claim che caratterizzano un'entità, insieme a metadati che permettono a chi riceve la credenziale, di ricostruire i dati trasmessi in un formato tale che permette di verificare l'autenticità, l'integrità e la validità dei dati contenuti.

```
1 {
2     "@context": [
3         "https://www.w3.org/2018/credentials/v1",
4         "https://www.w3.org/2018/credentials/examples/v1"
```

```
    ],
    "id": "179ea27d-8208-4215-81aa-594f9d0770d0",
    "type": "VerifiableCredential",
    "issuer": "ISSUER_DEVICE_0",
    "issuanceDate": "2021-10-26T12:44:56.519644",
    "expirationDate": "2022-10-26T12:44:38.732140",
    "credentialSubject": {
        "id": "did:ott:PNZLKKHUNKMLZAWXHQWPJJLZBDAI9SHQOJBVMXI9TUVKVSZRN9FJQRV
        ↪ QSDAKPF9RLICRAJUUNGXPRTQFQ",
        "type": "ApplicationType",
        "category": "SENSOR",
        "action": "READ"
    },
    "credentialSchema": {
        "id": "schema_id",
        "type": "JsonSchemaValidator2018"
    },
    "credentialStatus": {
        "id": "did:ott:I0...00#list#8",
        "type": "RevocationList2021Status",
        "statusListIndex": 8,
        "statusListCredential": "did:ott:I0...00#list"
    },
    "proof": {
        "type": "RsaSignatureSuite2018",
        "created": "2021-10-26T12:44:56.523641",
        "proofPurpose": "assertionMethod",
        "verificationMethod": "did:ott:I0...00#keys-2",
        "jws": "eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19..pLgf
        ↪ fIi...0DGSNy0Bg75DD9xbd9gqKlyRYiZiGRAuBZsNPSc-bay1JgXJHdN_D01ZCjdh
        ↪ qrYDAs48s2p1xh82UJyA"
    }
}
```

Listato 2.3: Esempio di Verifiable Credential in formato JSON.

2.3.2 Verifiable Presentation

Le Verifiable Presentation rappresentano la struttura dati con la quale un'entità trasmette le Verifiable Credential possedute [50] al fine di accedere ai servizi erogati dai Verifier.

Le Verifiable Presentation, come quella mostrata nel Listato 2.4, è quindi una struttura *wrapper* che contiene una Verifiable Credential (`verifiableCredential`), un identificativo (`id`) opzionale, un tipo (`type`), e una `proof`.

In questo caso, la proof è generata da un Holder. In essa quindi troveremo un riferimento alla chiave utilizzata per generare la signature dei dati contenuti nella presentazione, insieme agli altri campi che hanno la stessa semantica descritta nella sottosezione 2.3.1.

```
{
    "@context": [
        "https://www.w3.org/2018/credentials/v1"
    ],
    "type": "VerifiablePresentation",
    "verifiableCredential":
        {
            ...
            "id": "179ea27d-8208-4215-81aa-594f9d0770d0",
            ...
            "proof": {
                ...
                "jws": "eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il
                ↪ 19..pLgffIi...0DGSNy0Bg23OJC39jv9gqKlyRYiuBZsNPSc-bay1JgXJ
                ↪ HdN_D01ZCjdhqrYDAs48s2p1xh82UJyA"
            }
        },
    "proof": {
        "type": "RsaSignatureSuite2018",
        "created": "2021-10-26T12:45:64.528841",
        "proofPurpose": "assertionMethod",
        "verificationMethod": "PNZLKKHUNKMLZAWXHQWPJJLZBDAI9SHQOJBVMXI9TUVKVSZ
        ↪ RN9FJQRVQSDAKPF9RLICRAJUUNGXPRTQFQ#keys-2",
        "jws": "eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19..pLgf
        ↪ fIi...1ZCjdhqrYDAs69sd9xh82UE1A"
    }
}
```

Listato 2.4: Esempio di Verifiable Presentation in formato JSON.

2.3.3 Casi d'uso

Introdotti i modelli di Decentralized Identifiers, Verifiable Credential e Verifiable Presentation, in questa sezione verranno descritte le interazioni che intercorrono tra un Holder, un Verifier ed un Issuer. Lo scenario vede un Verifier che eroga un servizio in rete, raggiungibile attraverso l'endpoint specificato all'interno del suo DID document, associato ad un DID noto all'Holder, ed un Issuer che gestisce il rilascio delle credenziali. Vediamo quindi le azioni che dovrà intraprendere un peer Holder, dalla creazione della propria identità, fino alla revoca della credenziale utilizzata per l'accesso al servizio.

Creazione degli identificativi

La creazione dell'identità digitale parte dalla creazione degli Decentralized Identifier, insieme al materiale crittografico da inserire all'interno del relativo DID document. I tre

peer procedono quindi a generare due coppie di chiavi a crittografia pubblica e un DID che ricevono dal distributed ledger. In aggiunta, nel caso dell'Issuer, avviene la creazione di una status list che viene salvata nel Verifiable Registry del distributed ledger, in modo da essere accessibile a tutti gli altri peer della rete, e con cui gestirà lo stato delle credenziali. Il materiale privato, ovvero le chiavi reciproche di quelle pubbliche, vengono salvate in locale (nei nodi), mentre invece la corrispettiva parte pubblica viene inserita nel DID document, negli appositi campi `authentication` e `assertionMethod`, rispettivamente, la chiave per i processi di autenticazione e quella per la creazione delle proof.

Il document viene serializzato in un formato opportuno, inviato al distributed ledger e memorizzato sul registro. A questo punto, i peer sono pronti ad interagire.

Rilascio delle credenziali

Per accedere ad un servizio, e completare la propria identità digitale, l'Holder ha bisogno di farsi rilasciare una nuova credenziale da un Issuer. Prepara quindi un messaggio contenente tutte le informazioni necessarie a comporre il claim che sarà inserito nel `credentialSubject`.

Aperta la nuova connessione, Holder e Issuer iniziano un processo di mutua autenticazione, che avviene seguendo questi passi:

1. Scambio e lettura dei DID document dal distributed ledger.

2. Protocollo di challenge-response: vengono inviate delle challenge, cifrate utilizzando le chiavi pubbliche contenute nel campo `authenticationMethod` dei DID scambiati. Se il messaggio di sfida coincide con quello decifrato con le chiavi private di ognuno, allora il protocollo si conclude positivamente.

3. Generazione di un master-secret tramite scambio Diffie-Hellman: il segreto generato con le challenge sarà utilizzato per derivare delle chiavi di sessione con cui i messaggi saranno cifrati.

Completata l'autenticazione, i due peer instaurano un canale sicuro su cui avviene la comunicazione. L'Issuer allora rilascia la nuova credenziale, con una proof creata utilizzando la chiave privata reciproca di quella pubblica contenuta nel suo DID document, e la trasmette in maniera confidenziale all'Holder. Per verificarne la validità, l'Holder non dovrà fare altro che estrarre la chiave pubblica dal campo `assertionMethod` del DID document ricevuto dall'Issuer, ed analizzare la signature contenuta nella proof. Il caso d'uso si conclude quindi con il salvataggio in locale della credenziale.

Presentazione e accesso al servizio

L'Holder crea una nuova Verifiable Presentation, generando la proof contenente una signature creata con la sua chiave (privata), identificata dall'id contenuto nell'`assertionMethod` del suo DID document.

Recuperato il DID document del Verifier a cui accedere, l'Holder apre una connessione verso l'endpoint del servizio a cui vuole accedere, contenuto nel campo `services`. I peer

seguono lo stesso protocollo descritto nella sezione 2.3.3, e creano il canale sicuro su cui comunicare. Il Verifier allora, riceve la Verifiable Presentation, e ne verifica la proof utilizzando la chiave pubblica contenuta nell'`assertionMethod` del DID document ricevuto dall'Holder. Se le verifiche della proof e dei metadati della presentazione vanno a buon fine, il Verifier estrae la Verifiable Credential contenuta nella Verifiable Presentation, altrimenti segnala l'errore.

La proof della credenziale viene esaminata valutando la signature contenuta al suo interno, *senza* coinvolgere in alcun modo l'Issuer. È importante infatti notare, che l'unica informazione di cui ha bisogno il Verifier per procedere, è valutare se l'Issuer contenuto nel campo `issuer` sia fidato oppure no. Possiamo quindi definire l'insieme degli Issuer noti al Verifier, come una lista di peer verso cui viene posta una fiducia implicita, e che quindi prende il nome di trusted list.

Dopo aver effettuato la verifica della proof e dei metadati della credenziale, il Verifier procede all'elaborazione del `credentialSubject`. Le affermazioni espresse dai claim contenuti vengono utilizzati per valutare l'autorizzazione dell'Holder per accedere al servizio. Viene quindi elaborata una risposta per essere trasmessa sul canale sicuro creato in precedenza.

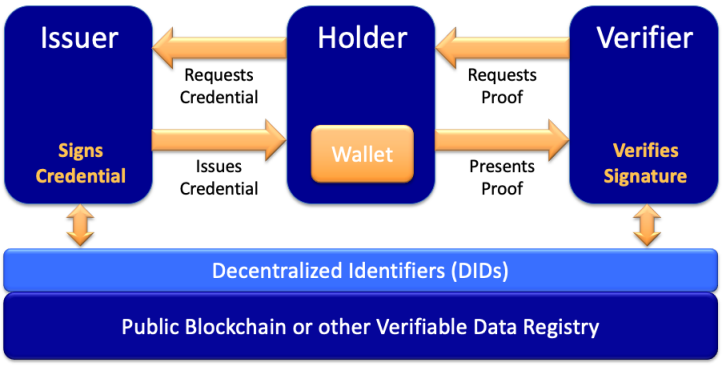


Figura 2.6: Interazioni tra i diversi tipi di peer. Immagine tratta da [46].

Revoca della credenziale

L'Holder ha la facoltà di poter richiedere la revoca di una credenziale rilasciata dall'Issuer. Seguendo quindi gli stessi passi descritti finora, che ricordiamo brevemente:

1. recupero del DID document dell'Issuer;
2. creazione di Verifiable Presentation;
3. autenticazione e creazione del canale sicuro con l'Issuer;

l'Holder include nella richiesta la credenziale da revocare. L'Issuer allora, recupera la status list dal distributed ledger e ne verifica lo stato corrente, valutando se sia il caso di procedere oppure no. Se infatti la credenziale è già stata revocata, verrà inviato un

messaggio di errore, altrimenti l'Issuer provvederà ad aggiornare la status list fornendo un messaggio di avvenuta operazione.

Inoltro di richiesta

In questo scenario sono coinvolti tre tipi di peer:

- Un Holder che richiede il servizio di inoltro.
- Un Relay che riceve la richiesta e inoltra il messaggio ad un Verifier.
- Il Verifier che riceve il messaggio inoltrato e produce una risposta diretta all'Holder.

Le connessioni tra Holder e Relay, e tra Relay e Verifier, avvengono seguendo le stesse procedure di autenticazione descritte in precedenza. Tuttavia, affinché Holder e Relay siano autorizzati a richiedere questo tipo di servizio, hanno bisogno di ottenere dall'Issuer una credenziale adeguata. Le richieste quindi, avvengono tramite lo scambio di Verifiable Presentation, dalla richiesta da parte dell'Holder, fino all'inoltro stesso verso il Verifier. In questo modo, ogni peer riesce a validare le richieste e quindi ad autorizzare l'accesso per ogni servizio richiesto.

2.3.4 Selective disclosure

Un protocollo Zero-Knowledge è un metodo attraverso il quale una parte, detta prover, riesce a verificare ad un'altra parte, detta verifier, che una proprietà è vera senza rivelare nient'altro che l'affermazione è vera. In altre parole, un'entità può autenticarsi ad un'altra senza rivelare l'informazione segreta. Tale meccanismo permette quindi ad un Holder di inviare parte di una Verifiable Credential, che il Verifier può verificare in zero-knowledge, trasmettendo solo una parte della credenziale.

La separazione che c'è tra i dati inviati e quelli su cui avviene la verifica inoltre, introduce un vantaggio legato agli schemi. Le credenziali rilasciate da un Issuer non dovranno per forza essere contenute in messaggi aventi lo stesso schema: un Verifier può richiedere ad un Holder, di inviare i dati in una forma che non ha gli stessi vincoli dei modelli utilizzati dall'Issuer. La selective disclosure è quindi l'abilità di un Holder di poter selezionare solo una parte delle proprie Verifiable Credential e di dimostrarne il possesso in zero-knowledge.

Il caso estremo della selective disclosure consente di utilizzare credenziali anonime, cioè di non rivelare *nessuna* delle informazioni contenute, ma di trasmettere dei dati che ne dimostrino il possesso. Questa verifica si basa sulla conoscenza di una signature, ovvero di una proof basata su un modello matematico che permette di dimostrare che l'Holder abbia una signature valida su un segreto, che coincide con l'intera credenziale (proof of knowledge of a signature). L'implementazione di tale meccanismo di verifica, si basa sulle CL signature [11], che prendono il nome da uno schema di firma ideato da Jan Camenisch e Anna Lysyanskaya.

2.4 Trust over IP

Trust over IP [27] è un progetto che propone un'architettura per rendere scalabile l'affidabilità digitale in Internet. Le componenti descritte nelle sezioni precedenti formano uno stack tecnologico che include le metodologie attraverso cui i peer della rete riescono a comunicare in sicurezza e nel rispetto della privacy. Ma ciò non basta, per rendere un sistema accettabile legalmente, è necessario definire le regole che rendono le Verifiable Credential valide anche dal punto di vista governativo.

Nel mondo reale esiste quello che viene definito il triangolo della fiducia, cioè quel triangolo formato dalle relazioni di fiducia implicite tra *chi verifica* le credenziali e *chi le rilascia*. I vertici di questo triangolo sono formati dai tre tipi di peer:

- Issuer: rappresenta la fonte delle credenziali.
- Holder: coloro che richiedono le credenziali agli Issuer, le salvano e le presentano ai Verifier.
- Verifier: chiunque effettui la verifica dell'autenticità e della validità delle credenziali presentate dagli Holder.

Affinché una credenziale sia veramente valida allora, occorre definire un secondo triangolo della fiducia che include la Governance Authority. Il compito di quest'ultima è quello di pubblicare uno standard chiamato Governance Framework, un documento che definisce le regole che i peer dell'ecosistema si impegnano ad osservare, e che autorizza alcuni dei nodi ad assumere un certo tipo di ruolo.

Nei primi anni dell'esplosione di Internet, il principale (se non l'unico) strumento per garantire gli accessi controllati era quello degli account: gli utenti ponevano la loro fiducia in una combinazione di username e password, il che li autorizzava ad ottenere l'accesso ad un server. Presto gli utenti aumentarono e nacquero i primi problemi di scalabilità. La soluzione fu quella di poter accedere a servizi diversi utilizzando un account singolo, di cui un'organizzazione terza faceva da garante: i federating account. Il problema principale dei single sign-on portal e del relativo modello di fiducia, è dovuto alla presenza degli intermediari. Ogni volta che un client deve accedere ad un servizio, c'è quindi bisogno di un secondo server.

La novità allora è quella di introdurre un *nuovo* tipo di fiducia peer-to-peer senza intermediari, per permettere alle entità interagenti di creare una relazione *diretta* con le altre, secondo regole definite da loro. Questo è esattamente il tipo di modello di fiducia che esiste nel mondo reale. Un soggetto infatti, è in grado di verificare e autenticare una controparte grazie alla fiducia implicita che pone in chi ha rilasciato l'informazione presentata, e in chi ne garantisce la validità. Trust over IP si pone quindi come obiettivo quello di realizzare lo stesso tipo di modello che esiste da sempre nel mondo reale, e di applicarlo al mondo digitale.

Decentralized Identifiers, Verifiable Credentials e Verifiable Data Registry sono le componenti che messe insieme formano lo stack tecnologico del Trust over IP. Il tassello mancante che serve per completare l'ecosistema è un secondo stack parallelo, quello di Governance.

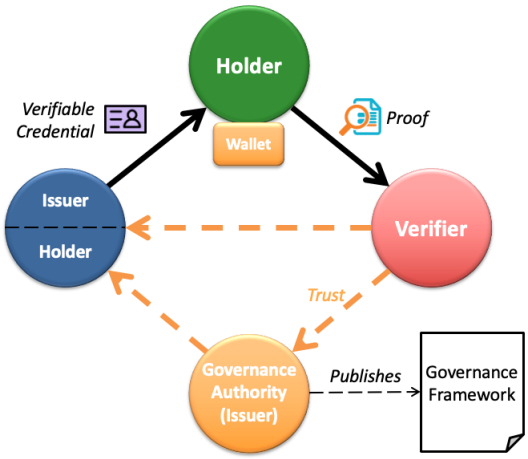


Figura 2.7: I due "triangoli della fiducia" per i tre diversi peer. Immagine tratta da [14].

2.4.1 Governance stack

Il Governance stack ha come obiettivo quello di rendere ogni livello dello stack tecnologico accettabile dal punto di vista governativo, legale e sociale [14]. Analizziamo quindi i vari livelli, descrivendo il modo in cui questo obiettivo può essere realizzato.

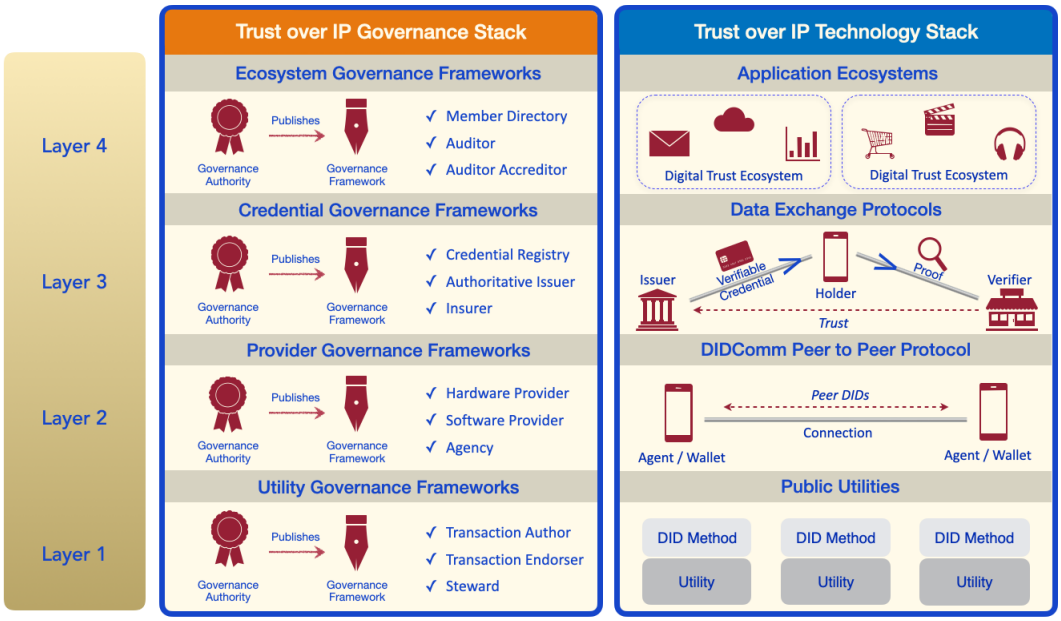


Figura 2.8: I livelli governativi e tecnologici del Trust over IP stack. Immagine tratta da [14].

Livello uno: utilità pubbliche

Alla base dello stack Trust over IP è presente la tecnologia che permette alle parti interagenti di poter creare canali sicuri e privati, utilizzando la crittografia pubblica. DID e DID document, combinati con la tecnologia dei ledger distribuiti creano una fonte di dati immutabili, la cui modifica è permessa solo a chi ne possiede il controllo. Ciò permette a tutti i peer di realizzare le verifiche crittografiche di cui hanno bisogno durante le loro interazioni.

Dal punto di vista governativo invece, il framework specifica le regole con cui le utility pubbliche sono implementate e come operano affinché le tecnologie sottostanti siano *affidabili* per i livelli sovrastanti.

Livello due: protocollo DIDComm

Il livello successivo riguarda il protocollo che definisce come due peer, tramite lo scambio dei DID, creano i canali sicuri. Gli agent/wallet rappresentano le repository degli Holder in cui avviene il salvataggio di tutto il materiale per le interazioni.

In questo livello, il Governance Framework garantisce le proprietà di privacy, sicurezza e protezione dei dati, che devono essere rispettate dai provider che offrono un'implementazione delle tecnologie dello standard DIDComm (autenticazione) e degli agent/wallet.

Livello tre: protocollo di scambio dei dati

Il livello tre è quello delle Verifiable Credential, con cui i peer possono verificare i dati che vengono presentati grazie al triangolo della fiducia introdotto nella sezione 2.4. Issuer, Verifier e Holder scambiano le credenziali attraverso i canali sicuri creati grazie al protocollo DIDComm.

Il governance framework definisce quindi quali sono i peer deputati al rilascio delle credenziali, cioè le entità di cui i Verifier si fidano implicitamente.

Livello quattro: ecosistema delle applicazioni

Il livello quattro è quello delle applicazioni che creano le interazioni basate sulla fiducia, per accedere a servizi di business, legali o sociali. Questo livello comprende gli ecosistemi digitali formati da tutte le applicazioni che condividono lo stack tecnologico e governativo composto dai livelli descritti.

Capitolo 3

Design

Nel seguente capitolo sarà illustrato il risultato dell'attività di progettazione. Le classi che compongono la struttura del framework verranno descritte mediante l'utilizzo di class diagram, mettendo in evidenza le interazioni che avvengono tra gli oggetti. Verranno inoltre giustificate le scelte che hanno portato alla definizione del design proposto, in relazione ai design pattern utilizzati.

3.1 Architettura

La caratteristica dei modelli di Decentralized Identifiers e Verifiable Credentials è quella di essere estensibili [47, 46]. L'obiettivo di questa prima fase di progettazione è quindi quello di creare un design modulare, che gode della stessa flessibilità dello standard, e di supportare l'introduzione di modelli nuovi [58].

Figura 3.1 illustra la struttura del framework suddivisa in moduli, le cui classi definiscono i comportamenti e le funzionalità dei peer. I moduli sono quindi suddivisi in:

- Model: classi che astraggono le entità e contengono le informazioni che le legano alle altre.
- Data Transfer Object (DTO): oggetti che incapsulano i dati che vengono trasmessi sulla rete.
- Handler: helper class utili a realizzare funzioni specifiche e comuni.
- Controller: espongono le API, ricevono le richieste dalla rete, le convertono nei DTO corrispondenti e orchestrano le classi dello strato Service per realizzare il caso d'uso.
- Service: contengono la logica di business e manipolano i Model.
- Repository: classi adapter che si interfacciano con lo strato di persistenza per salvare, recuperare o eliminare i Model.

Vediamo dunque come avviene, dal punto di vista generale, l'interazione tra le classi dei moduli appena descritti, quando un peer apre una nuova connessione e richiede un servizio.

Ipotizziamo che il dispositivo (nodo) che abbia già creato la propria identità, mediante i passaggi descritti nella sottosezione 2.3.3. Dopo aver aperto una nuova connessione verso un altro nodo, completa le procedure di mutua autenticazione e prepara il messaggio da inviare serializzandolo in formato JSON. La trasmissione avviene attraverso il canale sicuro, e il messaggio arriva al Controller del server che prende in carico la richiesta. La deserializzazione del messaggio viene completata creando un nuovo oggetto DTO, contenente tutte le informazioni che permettono ai livelli sottostanti di elaborare il tipo di richiesta. Gli Handler vengono quindi impiegati per integrare operazioni di verifica, per controllare l'accesso richiesto, e per gestire le condizioni di errore. In caso contrario infatti, il Controller risponderà con un messaggio di errore. I dati estratti dalla richiesta viaggiano verso lo strato dei Service, che si occupano di aggiornare lo stato dei Model servendosi delle Repository per accedere ai dati salvati. Se l'elaborazione è andata a buon fine, il Controller risponderà al nodo con il messaggio preparato dal Service.

3.2 Tipi di nodi e composizione delle classi

Siccome i dispositivi possono assumere più di un ruolo all'interno dell'ecosistema, essi implementano i comportamenti tipici dei nodi di rete: gli Holder possono essere considerati dei nodi client che aprono le nuove connessioni, verso gli altri nodi server a cui appartengono Issuer, Verifier e Tangle (distributed ledger). Vediamo quindi come attraverso la composizione delle classi del framework, si possono dare vita ai peer che realizzano i casi d'uso introdotti nel Capitolo 2.

3.2.1 Issuer

In riferimento alla Figura 3.2, un nodo Issuer dispone di un `CredentialController` che espone i due metodi per la gestione del rilascio e della revoca delle credenziali. Le richieste vengono accettate sotto forma di stringa insieme al `ConnectionContext`, contenente tutte le informazioni relative alla connessione aperta dall'Holder. La prima operazione sarà quella di trasformare la richiesta in un oggetto DTO compatibile, per poi procedere all'elaborazione e alla creazione della risposta.

In caso di rilascio (`handleIssueCredential`), l'Issuer si avvale del `CredentialService` per aggiornare la propria status list salvata sul Tangle. Il Tangle è raggiungibile grazie al `DidMethodHandler` che offre tutti i metodi necessari a creare le richieste ed ottenere i dati, sempre sotto forma di DTO.

La generazione di uno status da inserire nella nuova credenziale è affidato al `Creden⌋ tialStatusService` (`createCredentialStatus`), che in caso positivo si occupa dell'aggiornamento della status list (`issue`), mentre invece in caso negativo rifiuta la richiesta. La stessa procedura è valida per la revoca di una credenziale, la cui richiesta viene gestita dal metodo `handleRevokeCredential`.

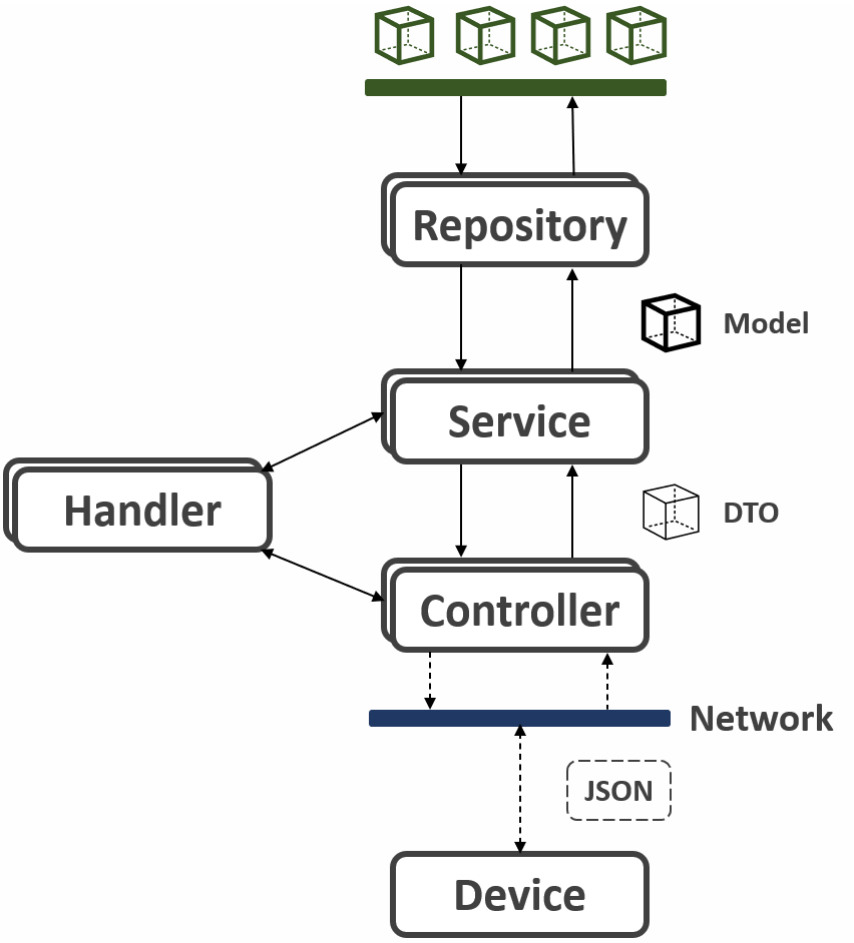


Figura 3.1: Moduli del framework.

Il `CredentialStatusService` si occupa anche di creare la status list (`createStatusLis t`), che sarà caratterizzata da un identificativo formato dal DID con cui l'Issuer è stato inizializzato. La forma con cui viene costruita è quella di una Verifiable Presentation che contiene un tipo particolare di Verifiable Credential definita Status List Credential. Per ogni operazione relativa alle credenziali gestite dall'Issuer, verrà salvata la copia aggiornata della status list sul Tangle.

La `getStatus` consente di recuperare lo stato corrente di una credenziale facendo un lookup sulla status list. Lo stato viene indicato da un numero intero: 0 se la credenziale è valida, 1 se la credenziale è stata revocata.

`IssuerService` gestisce invece le informazioni sugli Issuer, come DID, nomi e identificativi, e recupera le informazioni interfacciandosi con la `IssuerRepository`.

Al momento del rilascio di una nuova credenziale, l'Issuer deve creare una signature da inserire in una proof. Il `ProofService` offre la `createJwsSignature` che contiene la logica per generare la proof su un dato arbitrario, rappresentato da una mappa (`payload`).

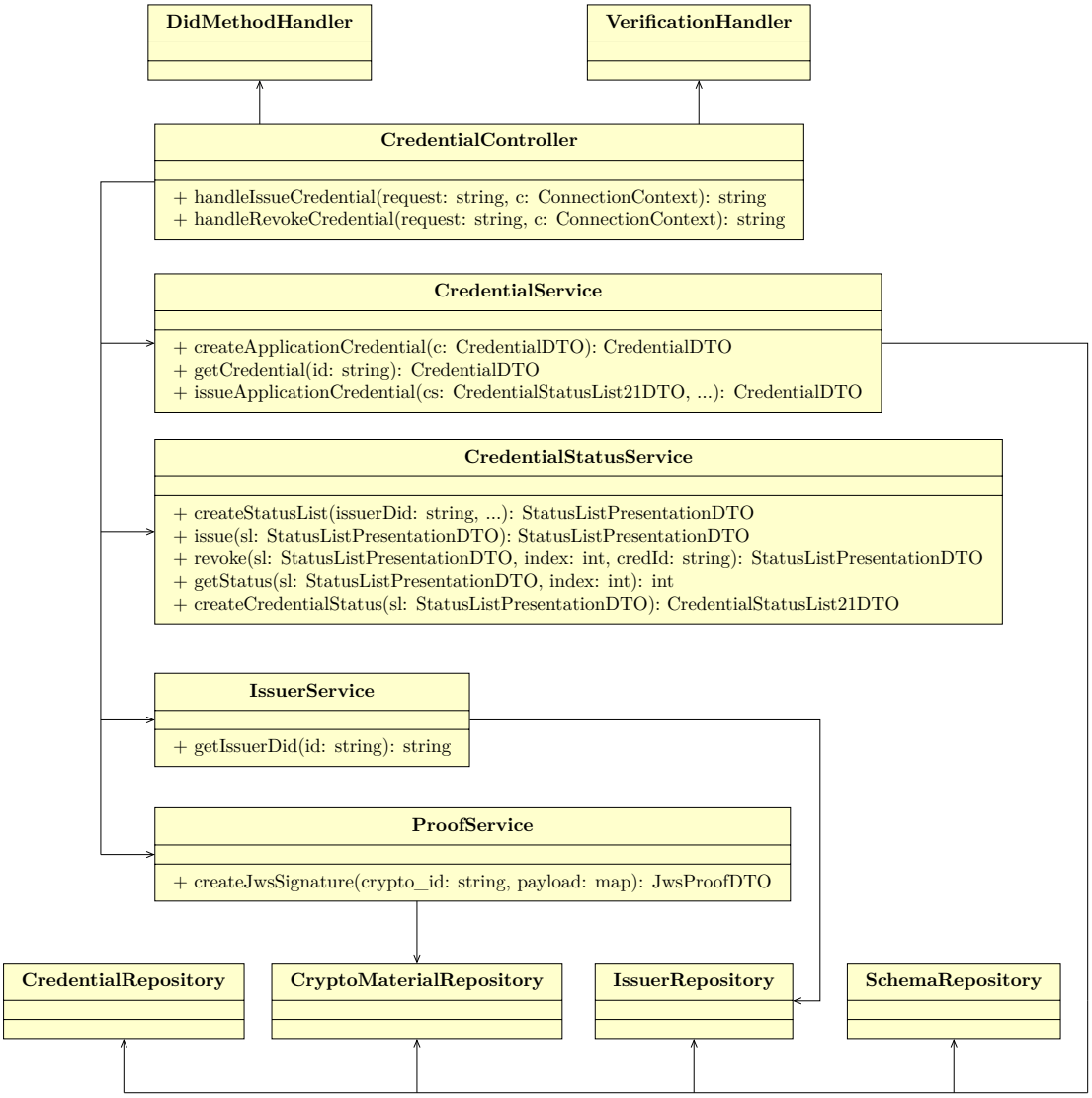


Figura 3.2: Diagramma delle classi per un nodo Issuer.

Utilizzando il materiale crittografico identificato da `crypto_id`, il Service crea la proof sotto forma di oggetto `ProofDTO`, che nel caso di signature JWS sarà `JwsProofDTO`.

Quando un Issuer riceve una nuova Verifiable Presentation da un Holder, provvede a verificare la richiesta tramite il `VerificationHandler`, composto dalle classi in Figura 3.3, che offre tutti i metodi necessari per validare metadati e proof

La `verifyPresentation` si occupa di validare una `PresentationDTO`, analizzando il DID document dell'Issuer, la status list corrente, e il contesto della connessione (`ConnectionC` ⌋ `ontext`). Se l'operazione va a buon fine, il estrae il `ClaimDTO` dalla credenziale, che servirà per il controllo dell'accesso. In caso contrario, sarà lanciata un'eccezione e l'Holder riceverà

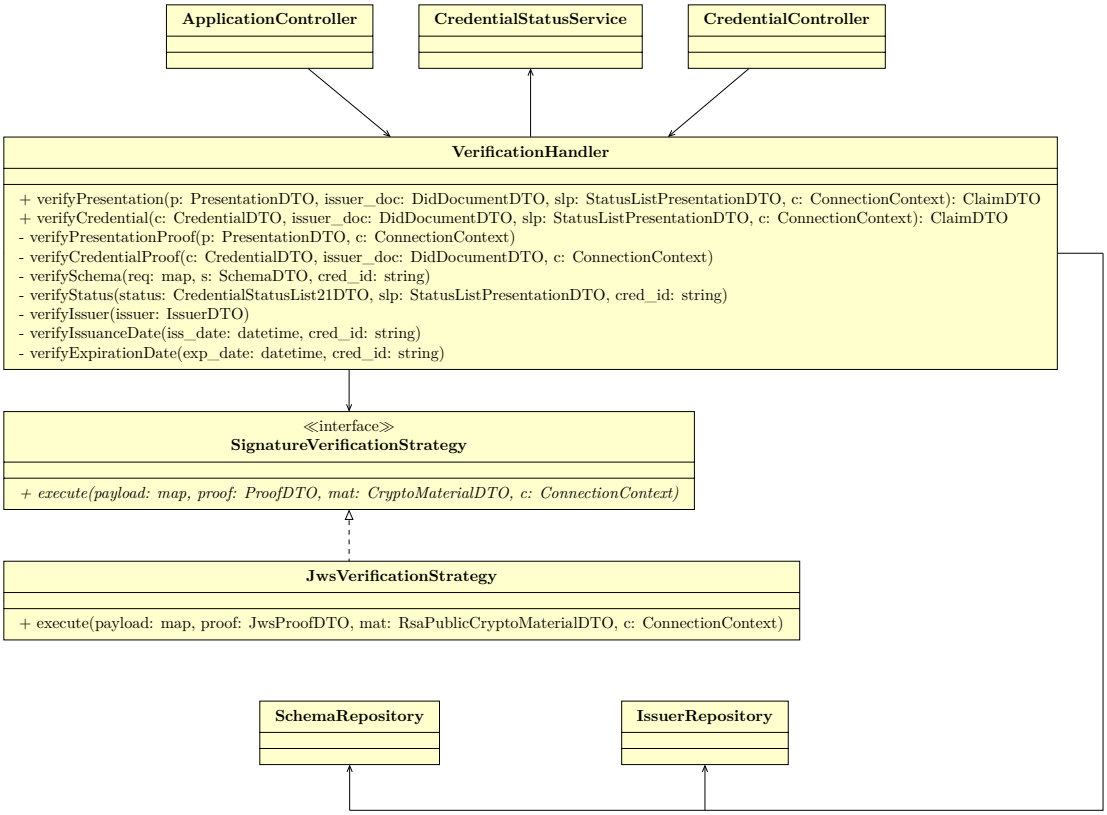


Figura 3.3: Diagramma delle classi per le funzioni di verifica.

un messaggio di errore.

La classe `SignatureVerificationStrategy` si occupa di validare la signature contenuta negli oggetti di tipo `ProofDTO`. Il dato su cui la verifica viene effettuata è passato sotto forma di mappa (`payload`), insieme al materiale crittografico `CryptoMaterialDTO` e al `C⌋onnectionContext`. L'implementazione concreta `JwsVerificationStrategy` è in grado di interpretare oggetti del tipo `JwsProofDTO` e di validare signature JWS. Il `VerificationHan⌋dler`, in pratica, utilizza le classi `SignatureVerificationStrategy`, senza curarsi del tipo di signature da verificare. Per questo motivo, anche il materiale crittografico atteso è di tipo generico. In altre parole, implementando classi del tipo `SignatureVerificationStrategy` è possibile supportare diversi tipi di proof secondo uno strategy pattern. Per rendere il `VerificationHandler` compatibile con nuovi tipi di signature, è sufficiente integrare nuove implementazioni concrete, come quella fornita dalla `JwsVerificationStrategy`.

3.2.2 Verifier e Relay

Il nodo Verifier, in Figura 3.4, rappresenta un peer che espone dei servizi, il cui accesso è determinato dalle Verifiable Credential presentate dagli Holder. Si tratta della tipologia di nodi più semplice, in quanto le uniche azioni da applicare alle Verifiable Presentation sono

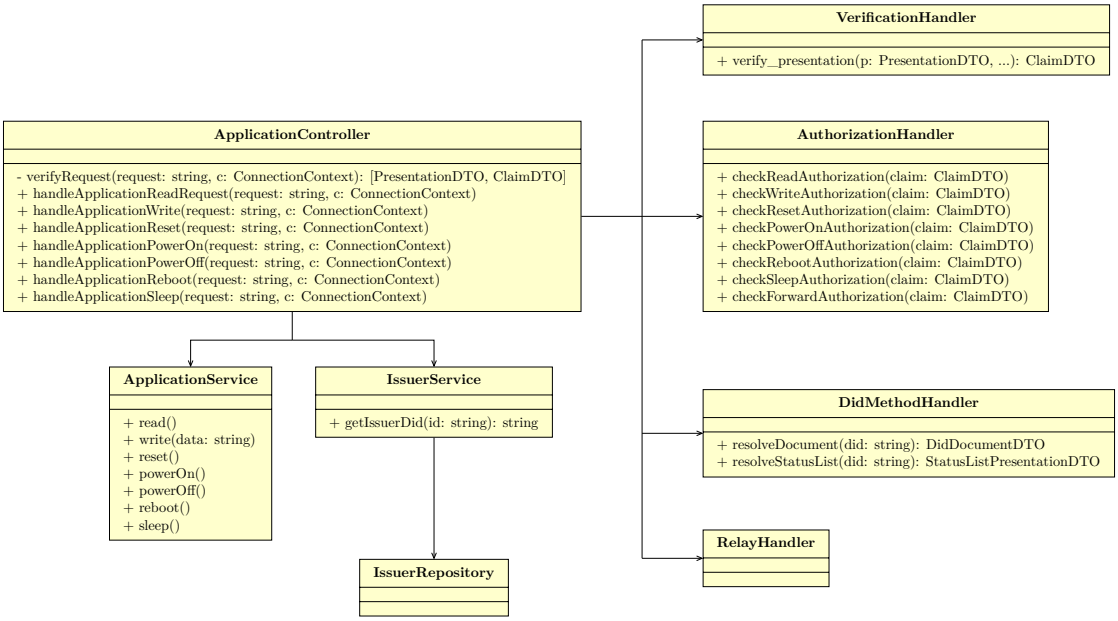


Figura 3.4: Diagramma delle classi per un nodo Verifier.

quelle di verifica. Infatti, per il Verifier le classi Handler che si occupano delle operazioni di validazione, rappresentano uno strato di sicurezza che protegge l'accesso alle funzionalità (cioè il servizio) che offre.

Le richieste vengono prese in carico dall'`ApplicationController`, che si occupa, come le altri classi di tipo Controller, della traduzione del tipo di dato in oggetto DTO. La `verifyRequest` applica gli stessi controlli visti nella sottosezione 3.2.1, ed estrae il claim della credenziale presentata dall'Holder.

Lo strato di servizio rappresentato da `ApplicationService` descrive le funzionalità offerte dal Verifier, accessibile solo se il claim inviatogli è compatibile con i privilegi dell'azione richiesta. `AuthorizationHandler` verifica infatti che l'Holder sia autorizzato ad accedere analizzando il `ClaimDTO` presentato. In caso positivo, l'accesso ai metodi dell'`ApplicationService` viene consentito, altrimenti viene mandato un messaggio di non autorizzazione.

Nel caso in cui un Holder richieda l'inoltro di una richiesta, è necessario includere nell'`ApplicationController` un oggetto del tipo `RelayHandler`. In questo scenario, il Verifier agisce da Holder verso un secondo Verifier, facendo da ponte per l'Holder che ha richiesto l'inoltro. La combinazione delle funzionalità di Holder e Verifier quindi, da vita ad un nuovo tipo di nodo denominato Relay. Per le funzionalità di rete, il Relay si serve del `ClientNetworkHandler`, che permette l'invio dei pacchetti verso gli altri nodi, agendo da Holder. Il `RelayHandler` costruisce quindi le Verifiable Presentation grazie al `PresentationService`, utilizzando i metodi per la generazione dei nuovi modelli. Il `VerifierService` invece, da accesso al pool di Verifier noti verso cui inoltrare la richiesta.

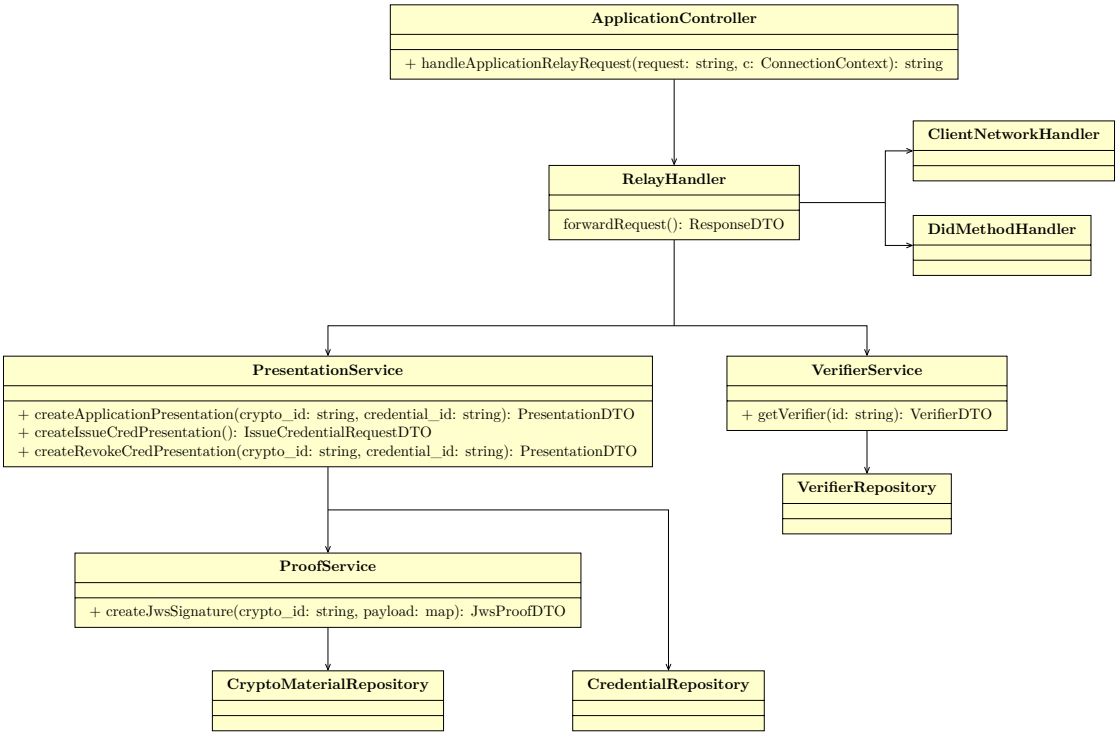


Figura 3.5: Diagramma delle classi per un nodo Relay.

3.2.3 Holder

Un nodo Holder si serve delle classi del modulo Service per salvare i Model ricevuti dagli altri nodi, e degli Handler per interagire con il Tangle e per aprire nuove connessioni.

Il `ClientNetworkHandler` fa parte del modulo di rete, che verrà descritto nella sezione 3.3. La classe offre un metodo `connect` per aprire le connessioni verso i nodi server, oppure per inviare un messaggio di ping e verificare che un nodo sia attivo.

`PresentationService` espone i metodi per la creazione delle Verifiable Presentation, mentre invece il `CredentialService` è utile per salvare una Verifiable Credential, quando viene ricevuta una nuova credenziale. Il `DidService` inoltre, è comune a tutti i nodi, poiché consente di recuperare il materiale crittografico e di creare un DID document, che sarà salvato sul Tangle tramite una richiesta creata dal `DidMethodHandler`. Infine, l'`IssuerService` e il `VerifierService` danno accesso ai pool dei peer noti, da cui vengono estratti i relativi DID e gli endpoint verso cui inviare le richieste.

3.2.4 Tangle

Il nodo fondamentale, che contiene il registro verificabile da cui tutti i peer ottengono le informazioni sugli altri peer, è il Tangle. In questo progetto, il (IOTA) Tangle rappresenta un'istanza concreta della classe DID Method, raffigurata in Figura 3.7, che si occupa di fornire in maniera sicura i Model salvati. L'interfaccia implementata definisce le operazioni

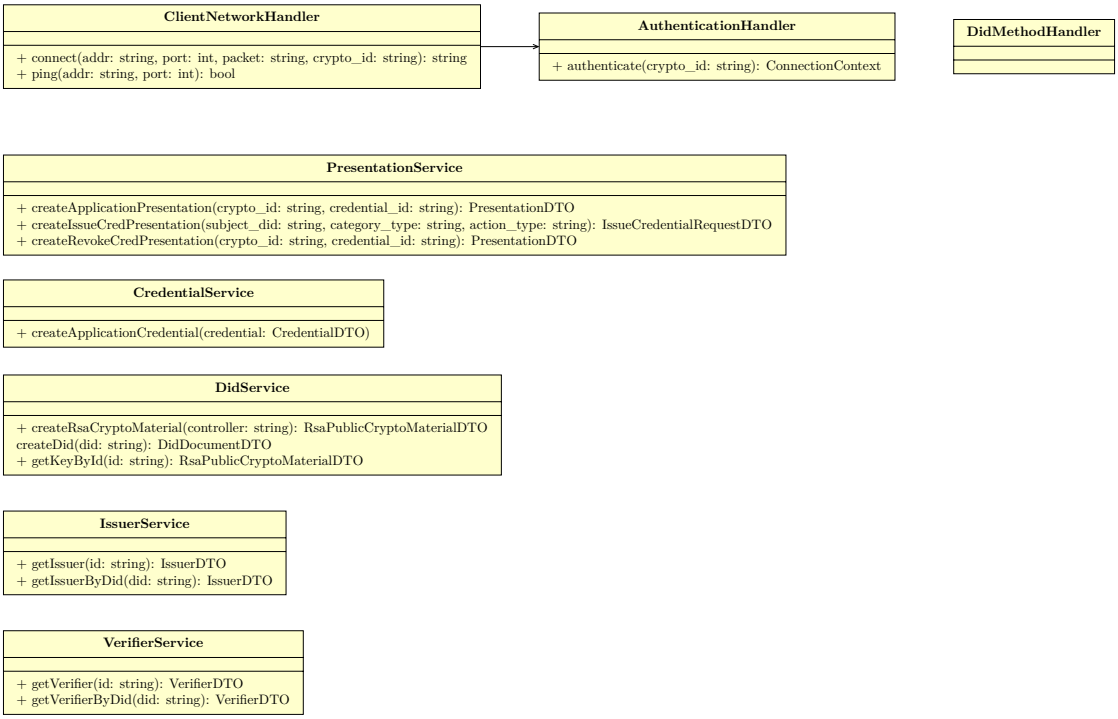


Figura 3.6: Classi utilizzate da un Holder.

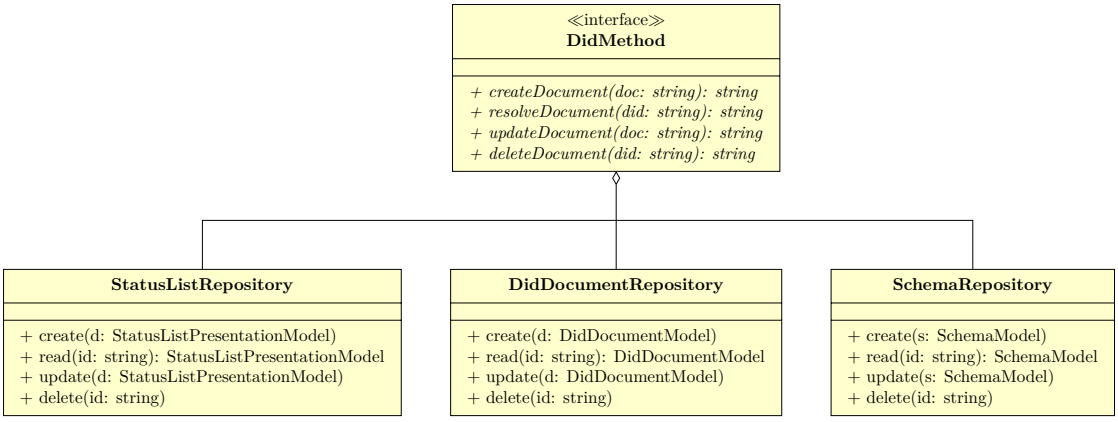


Figura 3.7: Classi che compongono il Tangle.

CRUD necessarie per gestire i DID document e si serve delle Repository per accedere ai Model. In particolare, la `StatusListRepository` gestisce le status list che sono accessibili da tutti i nodi dell'ecosistema, e lo `SchemaRepository` salva gli schemi che definiscono la struttura dei dati scambiati dai peer.

3.3 Funzionalità di rete

Affinché un oggetto di tipo Controller possa ricevere i messaggi dalla rete, occorre definire uno strato di rete *implicito*, che si occupa di gestire le connessioni. L'obiettivo è di creare degli oggetti che siano in grado di gestire le attese tra le richieste in maniera concorrente, e di chiudere le connessioni secondo dei timeout regolabili.

3.3.1 Descrizione del modulo

La logica per inizializzare un nuovo endpoint per il servizio, è quella contenuta negli oggetti `NetworkHandler`. La classe crea un'istanza di un server, che resta in ascolto su un indirizzo IP e una porta. L'inizializzazione permette di specificare una funzione che sarà invocata ad ogni connessione, cioè una `callback` della classe Controller in grado di servire il tipo di richiesta.

In Figura 3.8, vediamo che il `ServerNetworkHandler` si serve dell'`AuthenticationHandler` per eseguire il processo di autenticazione descritto nella sottosezione 4.2.2, tramite il metodo `authenticate`. L'implementazione concreta dell'handshake è contenuta in una delle classi derivate da `HandshakeMode`, il cui utilizzo segue la logica di uno strategy pattern. Facendo uso di un'interfaccia infatti, viene eliminata la dipendenza che c'è tra chi ne fa uso e chi ne fornisce l'istanza. `AuthenticationHandler` quindi, deve solo lanciare la `executeHandshake` per completare i giusti passaggi, relativi al tipo di nodo che esegue l'autenticazione.

Il risultato dell'handshake è il `ConnectionContext`, che contiene le seguenti informazioni:

- DID utilizzato per l'autenticazione.
- DID document utilizzato per la creazione delle signature.
- DID document del peer in comunicazione.
- Modalità di cifratura utilizzata per l'invio e la ricezione dei dati.

In altre parole, il `ConnectionContext` astrae il concetto di canale sicuro, su cui i nodi possono inviare e ricevere i dati in maniera confidenziale. I dati contenuti al suo interno rappresentano quindi lo stato della connessione tra due peer, che consente loro di poter *dialogare*.

Durante la fase di handshake vengono concordati i segreti per la derivazione delle chiavi di sessione. La logica per le operazioni crittografiche è specificata negli oggetti del tipo `CipherMode`, tramite i quali avviene la cifratura dei messaggi da scambiare. In questo tipo di oggetti, viene mantenuto lo stato della sessione tra i due peer, ovvero le chiavi di sessione che servono per garantire il requisito di confidenzialità. La procedura di derivazione delle chiavi quindi, risulta critica, e va configurata correttamente in base al tipo di nodo. Per questo motivo, la logica di creazione degli oggetti `CipherMode` avviene separatamente dal processo di derivazione.

Grazie all'utilizzo del pattern abstract factory, implementato nella classe `CipherModeFactory` in Figura 3.9, il peer può richiedere la creazione di una `CipherMode` di un tipo

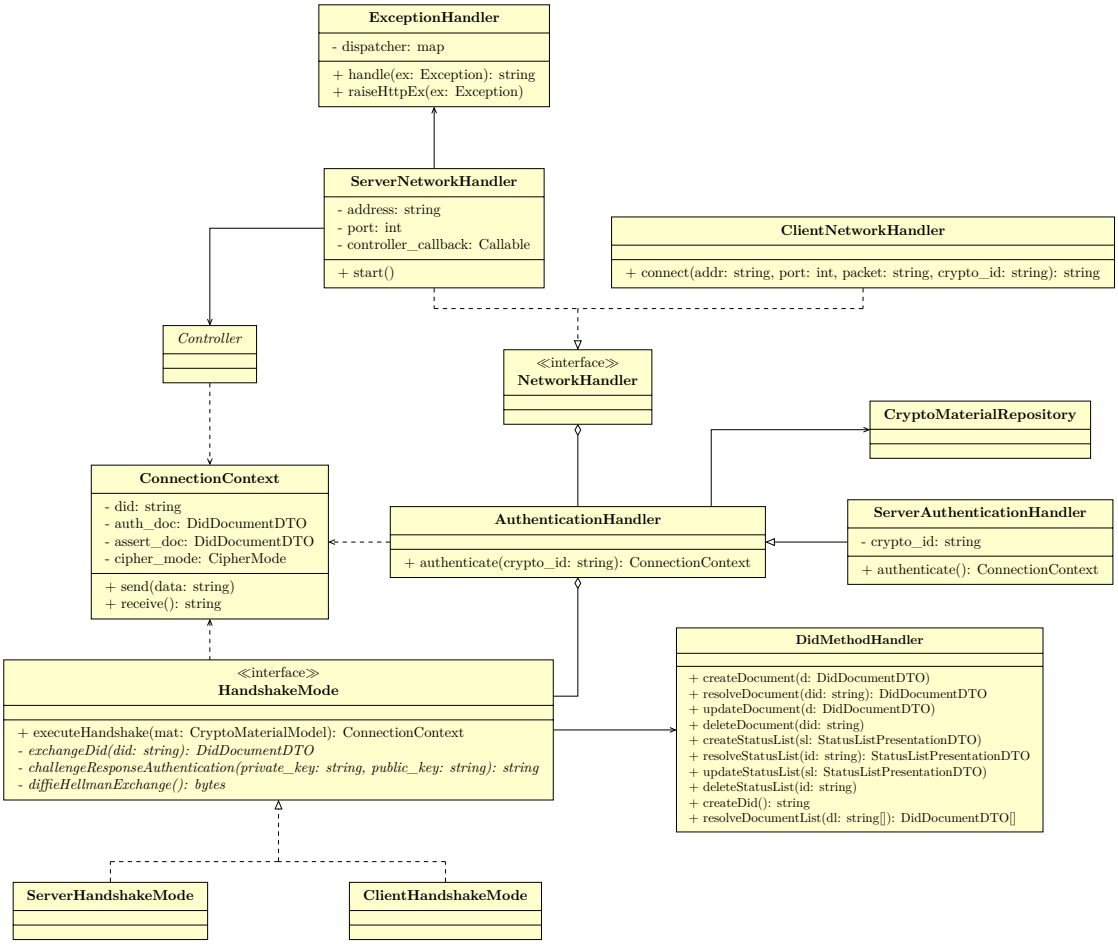


Figura 3.8: Diagramma delle classi che compongono gli endpoint e le componenti per l'autenticazione tra due peer.

`CipherModeType` specifico. `createCipherMode` si occupa quindi di fornire l'istanza della `CipherMode` secondo le modalità della factory utilizzata, in questo modo il master secret e il sale calcolati durante l'autenticazione, vengono inizializzati correttamente. Le classi `ServerCipherModeFactory` e `ClientCipherModeFactory` allora, implementano la logica per creare una `CipherMode` opportunamente configurata per derivare correttamente le chiavi di sessione.

La `CipherMode` creata, viene inserita nell'oggetto `ConnectionContext` che espone i metodi per l'invio dei messaggi sul canale sicuro. La `receive` legge il messaggio e lo decifra, mentre invece la `send` cifra il contenuto e lo spedisce.

Per ogni servizio offerto da Issuer o Verifier, ci sarà un endpoint composto dalle classi raffigurate in Figura 3.8. Il `ServerNetworkHandler` rimane in ascolto (`start`) delle nuove connessioni create con oggetti del tipo `ClientNetworkHandler` (`connect`). Gli Holder ad esempio, configurano i propri canali sicuri servendosi delle classi `ClientCipherModeFactory`.

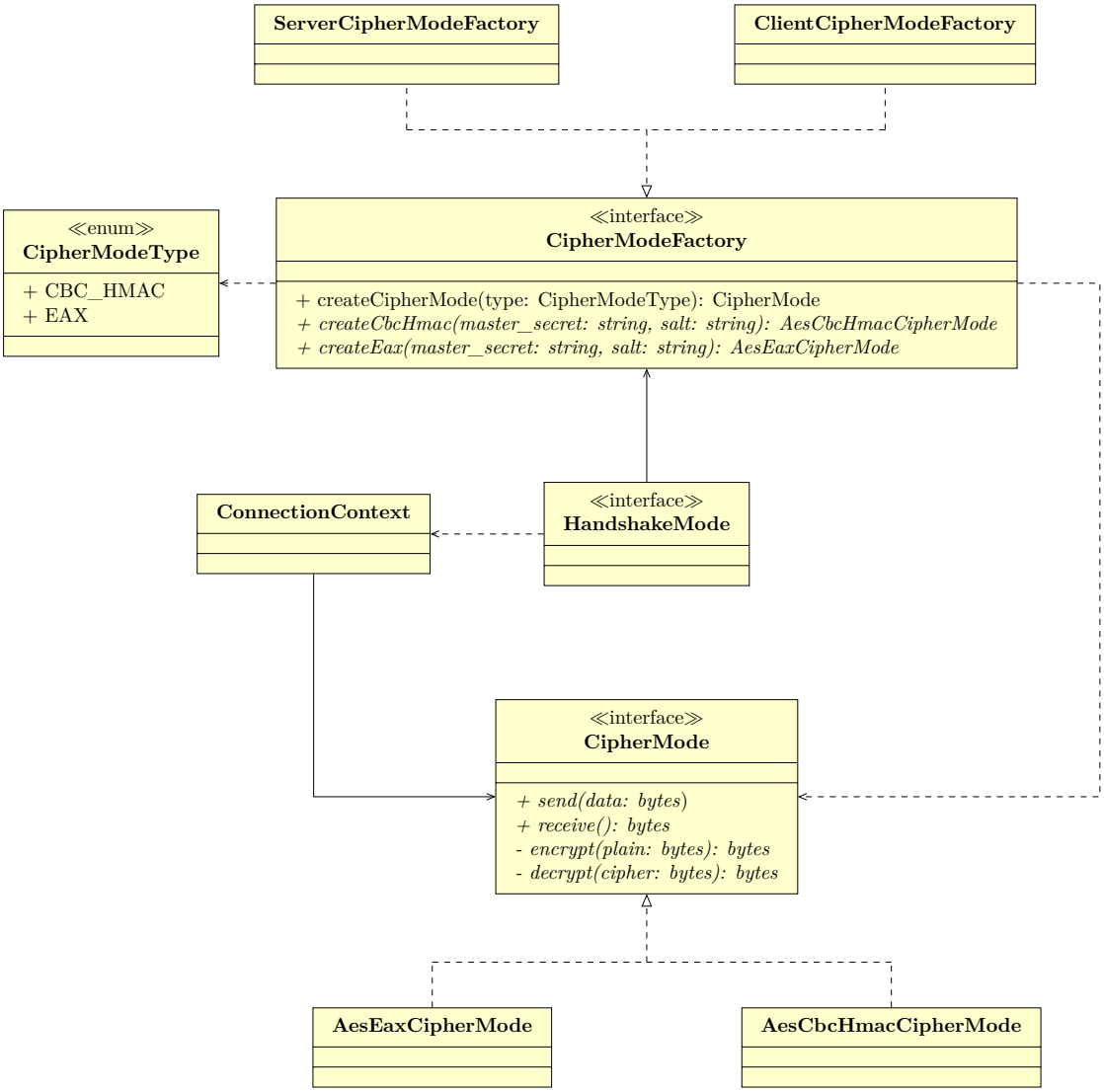


Figura 3.9: Diagramma delle classi che permettono di definire la modalità di cifratura, decifrazione e di autenticazione.

Infine, la classe `ExceptionHandler` viene utilizzata dal `ServerNetWorkHandler` per catturare tutte le condizioni di errore, tramite la `handle`. L'eccezione lanciata viene catturata per realizzare un mapping tramite il `dispatcher`, e generare una risposta per segnalare il tipo di errore.

3.3.2 Modelli e mapping

Oggetti del tipo `ProofModel` rappresentano solo un concetto astratto, che non pongono vincoli sul modello di Verifiable Credential. Il problema principale di un approccio simile è

legato alla serializzazione degli oggetti che viaggiano sulla rete. Solitamente un Controller in ascolto delle nuove richieste, già è a conoscenza del tipo di dato atteso, per cui l'operazione di mapping, che avviene alla ricezione, si riduce a quella di conversione di dati primitivi (solitamente stringhe) in oggetti DTO, e man mano che il dato viaggia verso i livelli sottostanti, subirà ulteriori conversioni fino a diventare un Model. Le complicazioni che possono nascere dalla serializzazione di tipi di dato variabili, trovano una possibile soluzione nel tenere separata la logica di conversione, dalle classi che descrivono la struttura che produce la conversione. Per questo motivo, il compito della classe `ModelMapper` è quello di trasformare gli oggetti in altre forme, ma di garantire che la mutabilità delle proprietà contenute siano rispettate. Il `ModelMapper` viene quindi utilizzato in maniera estensiva, per i dati in uscita e quelli in entrata, ma anche per ottenere le rappresentazioni necessarie alla generazione delle signature per i modelli.

Capitolo 4

Implementazione

In questo capitolo saranno affrontati gli aspetti implementativi del framework sviluppato. Saranno introdotte le librerie utilizzate per risolvere determinati aspetti del progetto, mettendo in evidenza le interazioni che avvengono tra le classi con l'ausilio di frammenti di codice. I casi d'uso saranno analizzati singolarmente, ponendo attenzione verso i modelli scambiati tra i nodi della rete e come i dati vengono elaborati nei vari livelli della struttura del codice.

4.1 Aspetti implementativi

Il Python è un linguaggio semplice, particolarmente adatto per lo sviluppo rapido di applicazioni. La libreria standard offre moduli per ogni tipo di esigenza, ma spesso non è sufficiente. Le primitive crittografiche ad esempio, essendo le operazioni più costose in termini di risorse, richiedono un'implementazione nativa (in C) delle funzioni, affinché siano completate in tempi ragionevoli. Per questo motivo, sono state adottate due librerie crittografiche molto diffuse, pycryptodome [33] e cryptography [32], le cui funzionalità sono state valutate analizzando le prestazioni offerte.

La scelta di affidarsi a moduli già pronti però, introduce problemi legati alle dipendenze, che hanno un certo peso sulla portabilità del codice. Nonostante ciò, le semplificazioni introdotte sono assai più convenienti rispetto alle risorse richieste, considerando anche i vantaggi che offrono in relazione agli sviluppi futuri del lavoro. Talvolta, queste saranno semplicemente delle interfacce verso le funzionalità già offerte dalla libreria standard del Python, ma incrementano la linearità e la chiarezza del codice, e di conseguenza, la manutenibilità dello stesso.

4.1.1 Gestione della persistenza

Data la natura dei modelli introdotti nel Capitolo 2, i vantaggi offerti dai database NoSQL (come MongoDB) si prestano perfettamente all'esigenza di non vincolare le informazioni a schemi relazionali. Le soluzioni open source disponibili non sono però compatibili con l'architettura target del progetto, la scelta è quindi ricaduta su sqlite3, che permette di

realizzare database efficienti e leggeri dal punto di vista computazionale [39], e di cui il Python fornisce un modulo standard. Per non creare un vincolo, l'utilizzo di sqlite3 sarà realizzato attraverso il toolkit SQLAlchemy [53], in modo tale da rendere il livello delle Repository compatibile con le API di altri tipi di database [54] e con driver aggiuntivi. Le Repository infatti, essendo delle classi adapter che fanno da interfaccia verso lo strato di persistenza, consentono di adottare soluzioni diverse da quelle dei database, semplicemente derivando nuovi tipi di implementazione. In questo modo, oltre ad ampliare le funzionalità del framework senza intervenire sulle vecchie classi, si possono riutilizzare gli stessi Model.

```python
@dataclass
class CredentialModel:

    _context: List[str]
    _id: str
    _type: str
    _issuer: IssuerModel
    _issuance_date: datetime
    _expiration_date: datetime
    _credential_subject: AppClaimModel
    _credential_schema: SchemaModel
    _credential_status: CredentialStatusList21Model
    _proof: ProofModel
    _owner: str
```

Listato 4.1: Classe che definisce il modello di Verifiable Credential (modulo `model`).

SQLAlchemy implementa classi che rappresentano i concetti del modello relazionale. Le operazioni di creazione di tabelle e di query sono incapsulate in oggetti che consentono di definirle in maniera programmatica, cioè senza utilizzare la sintassi delle istruzioni proprie dei database. Listato 4.2 mostra come attraverso i costrutti della libreria, si possono creare oggetti `Query`, che racchiudono le istruzioni SQL del dialetto con cui il motore di SQLAlchemy viene caricato. `select`, `where`, `insert`, `values`, hanno la stessa semantica delle istruzioni SQL, ma creano oggetti `Query` che vengono eseguiti utilizzando la `_conne⌋ ction` verso il database. Il risultato viene ottenuto come oggetto del tipo `CursorResult`, un puntatore alle `Row`, cioè alle righe della tabella di riferimento che soddisfano la query generata, e da cui possono essere estratti i modelli. Per ogni esecuzione allora, il motore di SQLAlchemy interpreta le operazioni costruite tramite gli oggetti `Query` e le traduce nelle istruzioni proprie del driver con viene caricato.

```python
def create(self, obj: object):
    # creazione di un oggetto Query
    stmt = select(self._table).where(self._table.c._id == obj._id)
    res: CursorResult = self._connection.execute(stmt)

```

```
6       if res.first() is not None:
7           raise ModelConflictException(
8               f"{obj.__class__.__name__} {obj._id} already exists in table
                ↪  {self._table.name}"
9           )
10
11      stmt = insert(self._table).values(obj.__dict__)
12      res = self._connection.execute(stmt)
```

Listato 4.2: Creazione di un nuovo modello nel database (classe `SqlAlchemyRepository`).

Il toolkit inoltre, offre funzionalità di Object-relational Mapping, un modulo costruito sul Core della libreria. L'ORM definisce un approccio in cui gli oggetti, che rappresentano i modelli, riflettono lo stato delle informazioni salvate. Tutte le operazioni sugli oggetti modello, vengono effettuate utilizzando direttamente i campi delle classi, le cui funzionalità vengono arricchite con quelle del modulo. I riferimenti che hanno verso gli altri modelli quindi, rappresentano le relazioni esistenti nello schema delle tabelle, e su cui vengono applicate le operazioni CRUD. Sarà SQLAlchemy che, in base alle operazioni invocate, esegue le istruzioni per aggiornare lo stato dei modelli.

Dato però l'overhead introdotto, e il numero di operazioni necessarie per inizializzare gli oggetti modello, si è scelto di utilizzare solo la libreria Core del toolkit, in modo da trarre vantaggio delle migliori prestazioni offerte durante l'esecuzione delle query [55].

La costruzione di una nuova tabella avviene attraverso l'oggetto `Table`, contenente le `Column` (i tipi di dato), le relazioni con le altre tabelle (chiavi esterne), e altri parametri di configurazione, come chiavi primarie. La definizione della tabella va associata ad un oggetto `MetaData`, contenente la collezione delle tabelle e dei relativi costrutti che saranno analizzati dal motore di SQLAlchemy per realizzare lo schema relazionale. Riferendoci al Listato 4.3, vediamo come avviene la costruzione di una tabella, mediante la funzione `create_engine` che crea un pool di connessioni, interfacciandosi con il driver specificato da `DB_URL`, cioè `pysqlite`, un driver per motori di database SQLite. La `create_all` sarà quindi invocata per eseguire le query di creazione con l'`engine`, che inizializzerà la collezione delle tabelle contenute nei `metadata`. Il risultato sarà la generazione di un file `.db`, che contiene il database con gli schemi definiti dagli oggetti racchiusi in `metadata`.

```
1   metadata = MetaData()
2   # definizione schema credential
3   credential_table = Table(
4       "credential",
5       metadata,
6       Column("_context", StringListType),
7       Column("_id", String, primary_key=True),
8       Column("_type", String),
9       Column("_issuer", String, ForeignKey("issuer._id")),
10      Column("_issuance_date", DateTime),
```

```
    Column("_expiration_date", DateTime),
    Column("_credential_subject", AppClaimType),
    Column("_credential_schema", ForeignKey("schema._id")),
    Column("_credential_status", CredentialStatusList21Type),
    Column("_proof", JwsProofType),
    Column("_owner", String),
)
...
DB_URL = "sqlite+pysqlite:///data.db"
# caricamento driver sqlite
engine = create_engine(DB_URL, ...)
metadata.create_all(engine)
```

Listato 4.3: Caricamento del driver sqlite3 e creazione dello schema della tabella `credential`.

Considerando i requisiti del progetto, alcuni modelli non sono necessari da indicizzare, ma solo da salvare. Quindi, piuttosto che creare altre tabelle e incrementare ulteriormente le relazioni tra gli schemi, è sufficiente definire nuovi tipi di dato che saranno interpretati da SQLAlchemy come tipi primari. L'esempio più semplice è dato dai `service` contenuti nei DID document, non c'è infatti interesse nel definire delle query attraverso le Repository per filtrare i documenti basati sui servizi. La soluzione è quindi quella mostrata nel Listato 4.4, che include il codice per la creazione di un nuovo tipo di Column estendendo la classe `TypeDecorator`. Il dato salvato, cioè la nuova colonna che va a definire, viene interpretata dal motore di SQLAlchemy come una stringa di tipo primitivo, specificato dalla proprietà `impl` inizializzata a `types.String`. Per gestire il flusso da e verso il database, è necessario specificare le operazioni che devono essere eseguite dal motore, per questo il `ServiceModel` al momento del salvataggio sarà convertito in una stringa JSON, mentre invece per la lettura avviene l'operazione inversa, cioè la ricostruzione del modello a partire dalla stringa.

```
class ServiceType(TypeDecorator):

    impl = types.String
    cache_ok = True

    def process_bind_param(self, value: ServiceModel, dialect):
        return json.dumps(value.__dict__)

    def process_result_value(self, value: str, dialect):
        model_dict = json.loads(value)
        return ServiceModel(**model_dict)
```

Listato 4.4: Definizione di un nuovo tipo di dato da inserire in una `Column`.

Vediamo quindi come lo strato Service fa uso dell'implementazione delle Repository per accedere ai dati. Listato 4.5 mostra la Repository astratta da cui derivano tutte le quelle che implementano le operazioni CRUD di base mediante SQLAlchemy. In alcune occasioni, questi metodi saranno sufficienti, è semplice infatti creare un oggetto Model partendo da un dizionario, oppure ricostruirli se non ci sono relazioni esterne, come mostrato nel Listato 4.6. Nei casi più complessi purtroppo, c'è bisogno di ridefinire la logica di un'intera operazione, ad esempio per modelli come `CredentialModel`, che contengono diversi riferimenti esterni. Nel ricostruire il modello quindi, è necessario dapprima ricostruire i modelli contenuti in seguito ad una lettura dalle altre tabelle, e quindi tramite le altre Repository.

```
1  class SqlAlchemyRepository(ABC):
2      def __init__(self, engine: engine, table: Table):
3          self._table = table
4          self._engine = engine
5
6      def __enter__(self):
7          self._connection: Connection = self._engine.connect()
8          return self
9
10     def __exit__(self, *args):
11         self._connection.rollback()
12         self._connection.close()
13
14     def commit(self):
15         self._connection.commit()
16
17     def create(self, obj: object):
18         stmt = select(self._table).where(self._table.c._id == obj._id)
19         res: CursorResult = self._connection.execute(stmt)
20         ...
21         stmt = insert(self._table).values(obj.__dict__)
22         res = self._connection.execute(stmt)
23
24     def read(self, id: str) -> dict:
25         stmt = select(self._table).where(self._table.c._id == id)
26         row: Row = self._connection.execute(stmt).first()
27         ...
28         return row._asdict()
29
30     def update(self, obj: object):
31         stmt = (
32             updt(self._table).where(self._table.c._id ==
                ↪  obj._id).values(obj.__dict__)
33         )
34         res: CursorResult = self._connection.execute(stmt)
35         if res.rowcount == 0:
36             raise RuntimeError()
```

```
37
38      def delete(self, id: str):
39          stmt = dlt(self._table).where(self._table.c._id == id)
40          res: CursorResult = self._connection.execute(stmt)
41          if res.rowcount == 0:
42              raise ModelNotFoundException(...)
```

Listato 4.5: Definizione della Repository base da cui derivare le altre specifiche per tipo di modello.

`SqlAlchemyRepository` racchiude la logica per creare la connessione verso la `table`, facendo uso dell'`engine`. Ogni Repository sarà quindi inizializzata con le tabelle contenenti i modelli di cui si occupano. Definendo inoltre i metodi `__enter__` e `__exit__`, gli oggetti del tipo `SqlAlchemyRepository` possono essere utilizzati tramite il Python context manager (costrutto `with`), che rende implicita l'esecuzione di alcune istruzioni, alla creazione e alla distruzione dell'oggetto Repository, come commit o chiusura della connessione.

```
1   class IssuerRepository(SqlAlchemyRepository):
2       def read(self, id: str) -> IssuerModel:
3           row = super().read(id)
4           return IssuerModel(**row)
5
6   class CredentialRepository(SqlAlchemyRepository):
7       def __init__(self, engine, cred_table, issuer_table, schema_table):
8           super().__init__(engine, cred_table)
9           self._issuer_table = issuer_table
10          self._schema_table = schema_table
11
12      ...
13      def read(self, id: str) -> CredentialModel:
14          stmt = select(self._table).where(self._table.c._id == id)
15          res = self._connection.execute(stmt)
16          cred_row: Row = res.first()
17          ...
18          if cred_dict["_issuer"] is None:
19              raise ModelNotFoundException(f"Credential {id} has no issuer")
20          ...
21          stmt = select(self._issuer_table).where(self._issuer_table.c._id ==
            ↪  issuer)
22          res = self._connection.execute(stmt)
23          issuer_row = res.first()
24          ...
25          issuer_model = IssuerModel(**issuer_dict)
26          ...
27          cred_dict["_issuer"] = issuer_model
28          ...
29          return CredentialModel(**cred_dict)
```

Listato 4.6: Lettura di un `CredentialModel` ricostruito con i modelli recuperati dalle altre tabelle (`IssuerModel`).

Grazie alla flessibilità di SQLAlchemy è possibile integrare tool esterni come SQLCipher.

SQLCipher [56] è un'estensione di SQLite che facilita la creazione di database cifrati facendo uso di librerie molto diffuse, come OpenSSL. La cifratura avviene in modo trasparente e *su richiesta*, le pagine vengono cifrate singolarmente tramite algoritmo AES a 256-bit in modalità CBC, con chiavi derivate da un algoritmo PBKDF2-HMAC-SHA512 a 256000 iterazioni (di default) partendo da un segreto e un sale. Ogni pagina è costruita con un vettore di inizializzazione (IV) casuale e con un Message Authentication Code (HMAC-SHA512), che vengono verificati ad ogni lettura dal disco. Questo comportamento permette infatti di rilevare se c'è stato un tentativo di modifica, oppure se è avvenuta una condizione di errore che ha causato la corruzione del database. In questi casi, SQLCipher arresta la connessione e segnala il problema.

Per integrare le funzionalità offerte da SQLCipher è sufficiente caricare il driver relativo senza fare alcuna modifica alle Repository, come mostrato nel Listato 4.7. La chiave di accesso viene specificata nell'istruzione `PRAGMA key`, prima delle istruzioni di creazione del database. Così facendo il file `.db` generato tramite SQLCipher apparirà come un file binario offuscato, a cui è possibile accedere solo conoscendo il segreto da cui è stata derivata la chiave di decifrazione.

Sebbene siano richieste operazioni aggiuntive per leggere e salvare le pagine su disco, l'overhead introdotto per le operazioni di cifratura e decifrazione ha un impatto minimo sulle prestazioni [57]. La soluzione quindi, è efficiente e non necessita di modifiche al codice delle Repository, che continuano ad interfacciarsi verso il database senza curarsi del driver utilizzato.

```
1    DB_URL = f"sqlite+pysqlcipher:///encrypted_{db_name}.db"
2    engine = create_engine(DB_URL, ...)
3    key_stmt = text(f"PRAGMA key = '{secret}';")
4    engine.connect().execute(key_stmt)
```

Listato 4.7: Inizializzazione di un database cifrato con SQLCipher.

4.1.2 Gestione delle richieste di rete

I dati ricevuti dalla rete vengono trasformati dai Controller in oggetti DTO (Data Transfer Object): semplici dataclass utilizzate per trasportare dati. Nel contesto di un software, i DTO vengono tipicamente adoperati come container per lo scambio di informazioni tra lo strato di presentazione e quello di business. Molti dei framework per applicazioni internet utilizzati sul mercato infatti, fanno un uso estensivo di questo approccio, che stabilisce

una sorta di *contratto* tra le informazioni ricevute dalla rete e quelle attese, andando a semplificare le operazioni di validazione e di serializzazione dei dati. I DTO contengono proprietà che mettono in evidenza le relazioni con le altre entità, e che sono necessarie allo strato di persistenza per ricostruire i modelli a cui fanno riferimento. Per questo motivo i campi contenuti al loro interno sono implementati utilizzando tipi primitivi, dunque già pronti per il processo di serializzazione. L'esempio riportato nel Listato 4.8, mostra una classe DTO che descrive una Verifiable Credential, contenente campi come `_issuer`, che includono un identificativo tramite cui è possibile leggere il relativo `IssuerModel` dal database.

```
@dataclass(frozen=True)
class CredentialDTO:

    _context: List[str]
    _id: str
    _type: str
    _issuer: str
    _issuanceDate: str
    _expirationDate: str
    _credentialSubject: AppClaimDTO
    _credentialSchema: SchemaDTO
    _credentialStatus: CredentialStatusList21DTO
    _proof: ProofDTO
```

Listato 4.8: Classe contenente le informazioni da trasmettere per una Verifiable Credential. Il parametro `frozen` rende una `dataclass` immutabile.

Siccome i modelli, e quindi i DTO a cui si riferiscono, devono poter essere verificabili in maniera crittografica, i passaggi di produzione e consumo (Figura 2.3) diventano critici. In questo contesto infatti, il concetto di rappresentazione assume una certa rilevanza, dovuta alla dipendenza che c'è tra il calcolo della signature e la rappresentazione del dato da validare. Un altro aspetto importante da tenere in considerazione, è quello dell'implementazione vera e propria dell'operazione di serializzazione. Python offre il modulo pickle, che però non è adatto per gli oggetti da trasmettere sulla rete, a causa di problemi di sicurezza [30]. La soluzione è stata quindi quella di sviluppare un `ModelMapper`, che ha la responsabilità di realizzare tutte le conversioni necessarie, nella rappresentazione JSON ad esempio, che definisce il modello canonico su cui avviene il calcolo delle signature e la validazione dei dati trasmessi.

L'implementazione, mostrata nel Listato 4.9, prevede la definizione di dizionari (`_model_dispatcher` ad esempio) che creano un mapping tra la classe dell'oggetto da convertire e la funzione incaricata della conversione, seguendo la stessa logica di uno strategy pattern. Per DTO semplici è sufficiente invocare la proprietà `__dict__` degli oggetti Python, contenente già un dizionario con i campi dell'oggetto. Per classi con molte relazioni invece, è necessario definire un metodo apposito.

```python
class ModelMapper:
    def __init__(self):
        self._model_dispatcher = {
            CredentialModel: self._cred_to_dto,
            PresentationModel: self._pres_to_dto,
            ...
        }

        self._dto_dispatcher = {
            CredentialDTO: self._cred_to_dict,
            PresentationDTO: self._pres_to_dict,
            ...
        }

        self._json_dispatcher = {
            CredentialDTO: self._json_to_cred_dto,
            PresentationDTO: self._json_to_pres_dto,
            ...
        }

    def to_dto(self, obj: object):
        return self._model_dispatcher[obj.__class__](obj)

    def to_json(self, obj: object) -> str:
        obj_d = self.to_dict(obj)
        return json.dumps(obj_d)

    def from_json(self, json_s: str, cls_type: Any) -> object:
        if json_s == "":
            return None

        json_d = json.loads(json_s)
        return self._json_dispatcher[cls_type](json_d)

    def to_dict(self, obj: object) -> dict:
        return self._dto_dispatcher[obj.__class__](obj)

    def from_dto(self, obj: object) -> object:
        return self._model_dispatcher[obj.__class__](obj)
```

Listato 4.9: Funzioni principali per la conversione degli oggetti.

Il `ModelMapper` è dunque fondamentale per i:

- Controller: la prima operazione, alla ricezione di una nuova richiesta, è quella di trasformare il messaggio ricevuto nella rappresentazione DTO attesa. In uscita

avviene il contrario, la risposta viene serializzata in una stringa partendo da un `ResponseDTO` che contiene i dettagli e lo status code.

- Service: quando un nuovo Model viene creato, occorre prima convertirlo nella forma canonica per calcolare la signature, dopodiché viene completato con la proof e salvato, oppure inviato come risposta.
- Handler: gli oggetti DTO vengono utilizzati per effettuare le verifiche preliminari per il controllo dei dati ricevuti.

Dopo aver esaminato la forma con cui i messaggi vengono trasmessi sulla rete, analizziamo *come* i nodi aprono le connessioni.

Il modulo Streams del Python [37] mette a disposizione primitive ad alto livello per lavorare con le connessioni di rete. Gli endpoint dei nodi server che espongono i servizi offerti, vengono costruiti con oggetti `ServerNetworkHandler`, la cui implementazione è riportata nel Listato 4.10. Il metodo `start` apre una socket all'indirizzo e alla porta specificate nel costruttore, che rimane in ascolto in maniera indefinita (`serve_forever`). Ad una nuova connessione, viene richiamata la callback registrata (`_handle_req`) che prende in carico la richiesta, generando due nuovi oggetti Stream, uno per le scritture (`StreamWriter`) e l'altro per le letture (`StreamReader`). Il primo passaggio che avviene prima della lettura della richiesta è quello di autenticazione, attraverso l'oggetto `AuthenticationHandler`, con cui è stato inizializzato l'endpoint. Il risultato del processo viene racchiuso nell'oggetto `ConnectionContext`, e passato alla callback del Controller che implementa la logica per servire la richiesta. Al termine dell'elaborazione, viene quindi inviata una risposta e il canale viene chiuso (`writer.close`).

```
class NetworkHandler(ABC):
    def __init__(self, name: str):
        self._name = name

class ServerNetworkHandler(NetworkHandler):
    def __init__(
        self,
        addr: str,
        port: int,
        c: Callable,
        name: str,
        ah: ServerAuthenticationHandler,
        ex: ExceptionHandler,
    ):
        super().__init__(name=name)
        self._address = addr
        self._port = port
        self._callback = c
        self._loop = asyncio.get_running_loop()
        self._ah = ah
        self._ex = ex
```

```
22
23      async def start(self):
24          try:
25              server = await asyncio.start_server(
26                  self._handle_req, self._address, self._port
27              )
28              addr = server.sockets[0].getsockname()
29              print(f"{self._name}: serving on {addr}")
30              async with server:
31                  await server.serve_forever()
32          except KeyboardInterrupt:
33              server.close()
34              server.wait_closed()
35              raise
36          except Exception:
37              raise
38
39      async def _handle_req(self, reader: StreamReader, writer: StreamWriter):
40          try:
41              conn_context: ConnectionContext = await self._ah.authenticate(
42                  reader=reader, writer=writer
43              )
44              req_s = await conn_context.receive()
45              try:
46                  res: str = await self._callback(req_s, conn_context)
47              except Exception as e:
48                  traceback.print_exc()
49                  res = self._ex.handle(e)
50              await conn_context.send(res)
51          except (asyncio.TimeoutError, asyncio.CancelledError):
52              err_print(self._name, "Client timeout")
53          except NetworkException:
54              traceback.print_exc()
55          except Exception:
56              traceback.print_exc()
57          finally:
58              writer.close()
```

Listato 4.10: Avvio e gestione delle connessioni di un nodo server (modulo `network`).

Un Holder invece, si serve dell'oggetto `ClientNetworkHandler` per l'apertura delle nuove connessioni, come mostrato nel Listato 4.11. Il messaggio (`packet`) viene inviato specificando la chiave da utilizzare per l'autenticazione (`crypto_id`), e mette il peer in attesa di una risposta per un tempo massimo, che definisce il timeout (`self._timeout`).

```
1   class ClientNetworkHandler(NetworkHandler):
2       def __init__(self, name: str, ah: AuthenticationHandler):
```

```
3           super().__init__(name)
4           self._ah = ah
5           self._timeout = 20
6
7       async def connect(
8           self, addr: str, port: int, packet: str, crypto_id: str
9       ) -> Union[str, None]:
10          res = None
11          try:
12              writer = None
13              coro = asyncio.open_connection(addr, port)
14              reader, writer = await asyncio.wait_for(coro, self._timeout)
15              conn_context: ConnectionContext = await self._ah.authenticate(
16                  reader=reader, writer=writer, crypto_id=crypto_id
17              )
18              await conn_context.send(data=packet)
19              res = await conn_context.receive()
20          except (asyncio.TimeoutError, asyncio.CancelledError):
21              err_print(self._name, "Server timeout")
22              raise ServerTimeout()
23          except NetworkException:
24              traceback.print_exc()
25          except Exception:
26              traceback.print_exc()
27          finally:
28              if writer is not None:
29                  writer.close()
30          return res
```

Listato 4.11: Metodo per aprire una nuova connessione verso un nodo server (modulo `network`).

Notiamo che in caso di errore, l'esecuzione passa al gestore delle eccezioni, un oggetto del tipo `ExceptionHandler`. Facendo riferimento al Listato 4.12, notiamo che l'implementazione segue la stessa logica del `ModelMapper`. Abbiamo infatti un dizionario contenente tutte le eccezioni definite nei moduli dell'applicazione, che vengono mappate con gli status code HTTP e un messaggio di dettaglio. Al verificarsi di un'eccezione, l'esecuzione converge fino allo strato di rete, che nella struttura dell'implementazione si trova al livello più alto. Per questo motivo, per ogni condizione di errore, l'`ExceptionHandler` si occupa di preparare una `ResponseDTO` adeguata con cui rispondere.

```
1   class ExceptionHandler:
2       def __init__(self, mm: ModelMapper, name: str = ""):
3           self._dispatcher = {
4               ValueError: ResponseDTO(
5                   _msg="", _code=HTTPStatus.UNPROCESSABLE_ENTITY
```

```
            ),
            ModelNotFoundException: ResponseDTO(
                _msg="", _code=HTTPStatus.NOT_FOUND
            ),
            ModelConflictException: ResponseDTO(
                _msg="", _code=HTTPStatus.CONFLICT
            ),
            AuthorizationException: ResponseDTO(
                _msg="", _code=HTTPStatus.UNAUTHORIZED
            ),
            ...
        }
        self._mm = mm
        self._name = name

    def handle(self, ex: Exception) -> str:
        err_print(self._name, str(ex))
        res = ResponseDTO(_msg="", _code=HTTPStatus.INTERNAL_SERVER_ERROR)
        try:
            res = self._dispatcher[ex.__class__]
        except Exception:
            traceback.print_exc()
        return self._mm.to_json(res)
```

Listato 4.12: Implementazione del gestore di eccezioni (modulo `handler`).

4.1.3 Concorrenza delle richieste

Creare un modulo di rete in grado di gestire le richieste in maniera concorrente, significa dover definire un meccanismo basato su thread multipli in ascolto. Seguendo tale approccio, è necessario fare un'attenta analisi su possibili race condition che si potrebbero creare per risorse locali e remote. Nel caso dei nodi server, ciò avrebbe portato ad ulteriori problemi dovuti al numero di thread da avviare, sia per garantire tempi ottimali nella ricezione delle risposte, sia per distribuire le risorse disponibili, già limitate sui dispositivi IoT. Tuttavia, a partire dalle versioni recenti del Python, è stato integrato un nuovo modulo che permette di scrivere codice asincrono, basato su un modello già proposto da tempo in altri ambienti di sviluppo web, come Node.js.

Il modulo asyncio [36] permette di scrivere codice asincrono in uno stile sincrono, secondo un approccio che si distingue da quello dei thread grazie all'utilizzo dei generatori, e al concetto di task, inteso come attività messa in esecuzione da uno scheduler. Il principale vantaggio in termini di prestazioni è dovuto all'eliminazione dell'overhead dovuto alla creazione e allo switch dei thread, che nel Python è particolarmente inefficiente a causa dei problemi introdotti dal GIL (Global Interpreter Lock) [29, 52] (Global Interpreter Lock), che limita l'efficienza del codice concorrente. I dispositivi IoT inoltre, non sono dotati di CPU potenti e multicore, da cui trarre vantaggio grazie al multiprocessing [12].

Asyncio rappresenta quindi lo stato dell'arte della programmazione concorrente in Python [29]. Il modulo esegue il codice concorrente in maniera *cooperativa*, gestendo lo scheduling delle operazioni tramite un Event Loop. Quando un'operazione ha bisogno di essere eseguita, viene creata una coroutine che viene messa in coda sullo scheduler dell'Event Loop. L'attesa per un risultato allora, viene avviata utilizzando la keyword `await` davanti all'invocazione della funzione asincrona. Se l'elaborazione del risultato richiede tempo, la funzione corrente è messa in pausa, lasciando che lo scheduler faccia uno switch. Non appena il risultato è pronto, questa viene risvegliata e l'esecuzione riprende dove era stata interrotta.

Se consideriamo ad esempio l'implementazione della funzione `_handle_req` nel Listato 4.10, possiamo notare che tutti i punti in cui una coroutine viene messa attesa con la keyword `await`, rappresentano i momenti in cui, il metodo che prende in carico le richieste di rete, lascia gestire l'esecuzione allo scheduler del `_loop`, permettendo ad un'altra coroutine sospesa di essere risvegliata. I nodi server quindi, servendo le richieste in maniera concorrente, riducono le attese dei client, che a loro volta possono aprire più connessioni contemporaneamente.

4.2 Casi d'uso

I casi d'uso presentati nelle sezioni seguenti vedono interagire quattro tipi di nodi:

- Issuer: espone i servizi per il rilascio e la revoca di credenziali digitali, gestendo le status list.
- Verifier: offre servizi accessibili dagli Holder con le credenziali digitali.
- Relay: fa da ponte tra Holder e Verifier.
- Holder: ottiene credenziali e accede ai servizi dei Verifier.
- Tangle: gestisce il Verifiable Registry su cui sono salvati i DID document e le status list.

4.2.1 Inizializzazione

Durante la fase di boot i peer creano un DID in maniera autonoma (Listato 4.13), generando le chiavi a crittografia pubblica da inserire nel relativo DID document (Listato 4.14). Una volta salvata la chiave privata nella propria Repository locale, il `DidDocumentModel` viene convertito nel DTO corrispondente per essere inviato al Tangle, il quale si occupa di ricostruire il modello dai dati trasmessi dall'Holder, e di salvarlo nel Verifiable Registry.

```
1 def create_did(self, did: str, ...) -> DidDocumentDTO:
2     auth_method_dto = self.create_rsa_crypto_material(controller=did, ...)
3     assert_method_dto = self.create_rsa_crypto_material(
4         controller=did, num=2, ...
```

```
    )
    ...
    doc = DidDocumentModel(
        _id=did,
        _authentication_method=auth_method,
        _assertion_method=assert_method,
        ...
    )
    return self._mm.to_dto(doc)
```

Listato 4.13: Codice per la creazione del materiale crittografico da inserire in un DID document (classe `DidService`).

Le chiavi vengono generate dal modulo RSA della libreria cryptography e inserite in un modello unico `RsaCryptoMaterialModel`, salvato nella `crypto_repository`. La proprietà `controller` determina il possessore del materiale creato, mentre invece gli identificativi vengono generati partendo dal DID del peer e aggiungendo un fragment per distinguere le chiavi e renderle univoche. Il risultato è un oggetto `RsaPublicCryptoMaterialDTO` che viene inserito in un `DidDocumentDTO` e inviato al Tangle.

```
def create_rsa_crypto_material(self, controller: str, ...,) ->
↪   RsaPublicCryptoMaterialDTO:
    _id = f"{controller}#keys-{num}"
    _controller = controller
    ...
    _private_key_pem = private_key.private_bytes(...).decode("utf-8")
    _public_key_pem = (private_key.public_key()...)
    pub = RsaPublicCryptoMaterialModel(
        _id=_id,
        _type=CryptoMaterialType.RSA.value,
        _controller=_controller,
        _public_key_pem=_public_key_pem,
    )
    pri = RsaPrivateCryptoMaterialModel(_private_key_pem=_private_key_pem)
    mat = RsaCryptoMaterialModel(
        _id=_id,
        _type=CryptoMaterialType.RSA.value,
        _controller=_controller,
        _private_material=pri,
        _public_material=pub,
        ..
    )
    with self._crypto_repository:
        self._crypto_repository.create(mat)
        self._crypto_repository.commit()
    return self._mm.to_dto(pub)
```

Listato 4.14: Codice per la creazione di chiavi RSA (classe `DidService`).

Rispetto agli altri peer, l'Issuer segue un passaggio in più, quello di creazione della status list (Listato 4.15). Lo `StatusListPresentationModel` viene composto definendo dapprima uno `StatusList2021ClaimModel`, contenente una stringa, che viene generata codificando e poi comprimendo una sequenza di bit, cioè una bitmap. La funzione `get_empty_encoded_list` ne genera una con tutti i bit inizializzati a zero e di dimensione `n`, da inserire nel claim, che viene quindi completato con un identificativo composto dal DID dell'Issuer e da un fragment. Il `StatusList2021ClaimModel` viene allora inserito in una `StatusListCredentialModel` e a sua volta in una `StatusListPresentationModel`, entrambi firmati dall'Issuer utilizzando il materiale identificato da `crypto_id`. Il risultato assume la forma di un oggetto `StatusListPresentationDTO`, che sarà salvato sul registro del Tangle. Come per i DID document, anche la status list può essere letta da tutti i peer dell'ecosistema tramite l'operazione di resolve.

```
def create_status_list(self, issuer_did: str, crypto_id: str, ..., n: int = 8)
    -> StatusListPresentationDTO:
    empty_encoded_list = get_empty_encoded_list(num=n)
    status_list_did = f"{issuer_did}#list"
    status_claim = StatusList2021ClaimModel(
        _id=status_list_did,
        _type=ClaimType.STATUS.value,
        _encoded_list=empty_encoded_list,
    )
    status_list_cred = StatusListCredentialModel(
        _issuer=issuer_did,
        ...,
    )
    ...
    proof_cred_dto =
    self._proof_service.create_jws_signature(crypto_id=crypto_id, ...)
    proof_cred = JwsProofModel(
        ...,
        _verification_method=crypto_id,
        _jws=proof_cred_dto._jws,
    )
    status_list_cred._proof = proof_cred
    status_list_pres = StatusListPresentationModel(
        _verifiable_credential=status_list_cred,
        ...
    )
    ...
    proof_pres_dto =
    self._proof_service.create_jws_signature(crypto_id=crypto_id, ...)
    proof_pres = JwsProofModel(
```

```
        ...
        _verification_method=crypto_id,
        _jws=proof_pres_dto._jws,
    )
    status_list_pres._proof = proof_pres
    return self._mm.to_dto(status_list_pres)
```

Listato 4.15: Codice per l'inizializzazione di una nuova status list (classe `CredentialStatusService`).

L'inizializzazione della trusted list dovrebbe essere un passaggio manuale. Tuttavia, per convenienza, si sono definite delle funzioni per creare un pool di Issuer e di Verifier noti, come mostrato nel Listato 4.16, che danno la possibilità all'Holder di recuperare, attraverso la lettura del DID document, l'indirizzo dell'endpoint su cui vengono erogati i servizi.

```
async def create_issuer_pool(issuer_service: IssuerService, pool_size: int) ->
↪  List[Tuple[str, str]]:
    issuer_pool = []
    for i in range(pool_size):
        try:
            issuer_id = f"ISSUER_DEVICE_{i}"
            issuer_did = "did:ott:I" + f"{i}" * 80
            issuer_name = f"ISSUER_{i}"
            _ = issuer_service.add_issuer(
                id=issuer_id, did=issuer_did, name=issuer_name
            )
        except ModelConflictException:
            pass
        issuer_pool.append((issuer_id, issuer_did))
    return issuer_pool
```

Listato 4.16: Codice per la creazione di un pool di Issuer noti.

A questo punto Issuer e Verifier rimangono in ascolto, mentre l'Holder è pronto per aprire nuove connessioni verso i nodi server.

4.2.2 Autenticazione e canali sicuri

La fase di autenticazione, che precede l'accesso al servizio, è composta dai seguenti passaggi:

1. Scambio dei DID: ogni peer invia il proprio DID, per permettere all'altro di leggere dal Tangle il DID document corrispondente, ed estrarre le chiavi a crittografia pubblica, necessarie per i passi successivi.

2. Protocollo challenge-response (Figura 4.1): il client genera una stringa casuale, che cifra con la chiave pubblica del server e invia. Il server riceve la challenge e la decifra con la sua chiave privata reciproca, dopodiché invia a sua volta una risposta che comprende una challenge cifrata con la chiave pubblica del client e la challenge ricevuta (decifrata). Dopo che il client risponde con la challenge del server decifrata con la sua chiave privata, entrambi i peer verificano che le risposte ricevute coincidono con quelle inviate. In caso positivo viene creato un sale con le challenge decifrate, altrimenti viene lanciata un'eccezione e la connessione viene interrotta.

3. Scambio Diffie-Hellman: il protocollo viene seguito per scambiare un master secret, che combinato con il sale del passo precedente, sarà utilizzato come input per una hashed key derivation function che crea le chiavi di sessione.

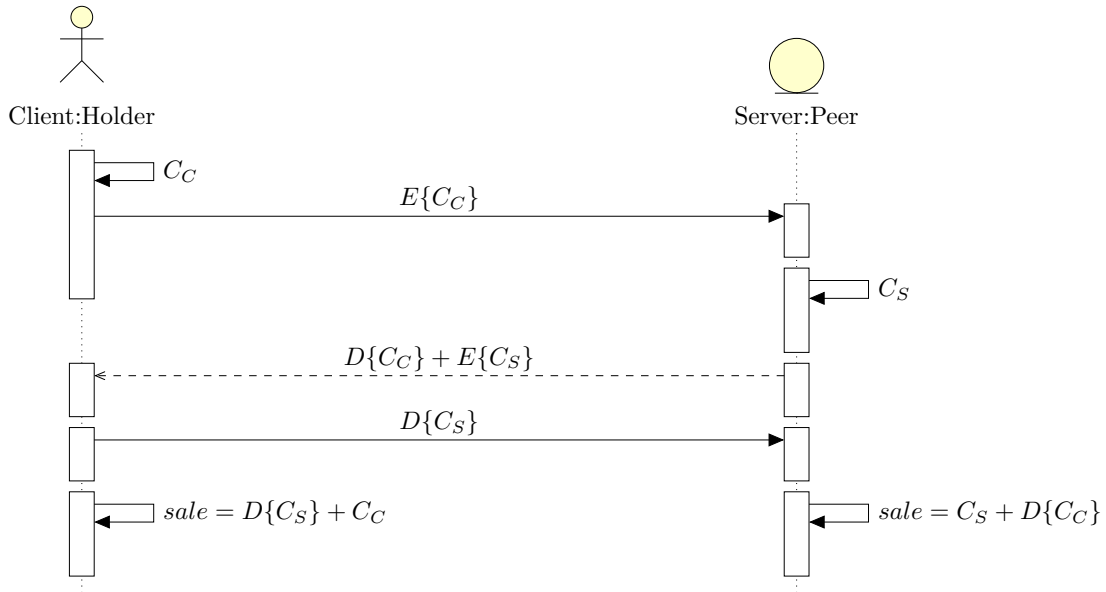


Figura 4.1: Passi di challenge-response per il protocollo di autenticazione.

La classe `HandshakeMode`, contenente la logica per il processo di autenticazione (Listato 4.17), viene inizializzata con un oggetto factory del tipo `CipherModeFactory`, che viene utilizzato alla fine dell'handshake per derivare la `CipherMode` da inserire nel `ConnectionC` ⌋ `ontext`. Il metodo factory `create_cipher_mode` viene invocato specificando il `cipher_type`, che determina il tipo dell'istanza richiesta. La `CipherMode` viene quindi inizializzata con le chiavi correttamente derivate, in base al tipo di factory invocata.

```
async def execute_handshake(
    self, reader: StreamReader, writer: StreamWriter, mat:
    ↪  RsaCryptoMaterialModel
) -> ConnectionContext:
```

```
    did = mat._controller
    received_doc: DidDocumentDTO = await self._exchange_did(did=did, ...)
    received_pub_key = received_doc._authenticationMethod._publicKeyPem
    ...
    salt_string = await self._challenge_response_authentication(
        private_key=rsa_private_key,
        public_key=rsa_public_key,
        ...
    )
    master_secret = await self._diffie_hellman_exchange(...)
    cm = self._cipher_mode_factory.create_cipher_mode(cipher_type=self._cm_typ
    ↪ e)(
        master_secret=master_secret,
        salt_string=salt_string,
        ...
    )
    return ConnectionContext(did=did, auth_doc=received_doc, cm=cm)
```

Listato 4.17: Logica per il processo di autenticazione comune a client e server.

Consideriamo l'esempio del Listato 4.18, in cui alla fine dell'handshake viene richiesta la creazione di un oggetto `AesEaxCipherMode`. In base alla versione del metodo `create_eax` invocata, l'oggetto risultante assume la configurazione specificata dal tipo concreto di `CipherModeFactory`. Grazie a questo pattern, entrambi i peer saranno allora in grado di costruire un `ConnectionContext` corretto. Lo stato della connessione sarà quindi descritto dalle chiavi di sessione derivate, i DID scambiati durante l'handshake, e gli oggetti Stream che invocano le primitive di sistema per l'invio e la ricezione dei dati.

```
    class CipherModeFactory(ABC):
    def __init__(self):
        self.dispatcher = {
            CipherModeType.CBC_HMAC: self.create_cbc_hmac,
            CipherModeType.EAX: self.create_eax,
        }

    def create_cipher_mode(self, cipher_type: CipherModeType) -> CipherMode:
        return self.dispatcher[cipher_type]

    @abstractmethod
    def create_eax(
        ...
        master_secret: bytes,
        salt_string: str,
    ) -> AesEaxCipherMode:
        pass

```

```
    ...
    # client
    def create_eax(
        ...
        master_secret: bytes,
        salt_string: str,
    ) -> AesEaxCipherMode:
        keys = key_derivative_function(master_secret, 32, salt_string, 2)
        return AesEaxCipherMode(
            reader=reader, writer=writer, enc_key=keys[0], dec_key=keys[1])

    # server
    def create_eax(
        ...
        master_secret: bytes,
        salt_string: str,
    ) -> AesEaxCipherMode:
        keys = key_derivative_function(master_secret, 32, salt_string, 2)
        return AesEaxCipherMode(
            reader=reader, writer=writer, enc_key=keys[1], dec_key=keys[0]
        )
```

Listato 4.18: Creazione di istanze di `AesEaxCipherMode` correttamente configurate per l'utilizzo da parte di un client o di un server.

4.2.3 Gestione della status list

Le status list gestite dagli Issuer sono contenute all'interno di strutture dette Status List Presentation (Listato 4.19), che a loro volta contengono le Status List Credential. Una Status List Credential è un tipo particolare di Verifiable Credential in cui viene inserita una stringa codificata di una bitmap compressa (Figura 4.2). Ogni bit della bitmap contiene lo status di una credenziale: 0 corrisponde a credenziale valida, 1 a credenziale revocata. Verificare lo status di una Verifiable Credential significa quindi eseguire un lookup nella bitmap ed ispezionare il suo bit riservato.

Consideriamo la Verifiable Credential nel Listato 2.3, e notiamo che il campo `credentialStatus` include al suo interno le seguenti proprietà [20]:

- `statusListIndex`: indice del bit all'interno della bitmap.
- `statusListCredential`: id (un DID) che identifica la Status List Presentation.
- `id`: combinazione delle due proprietà precedenti.

Per ogni aggiornamento sullo stato delle credenziali, l'Issuer:

1. Ricalcola le proof di Status List Presentation e Status List Credential.

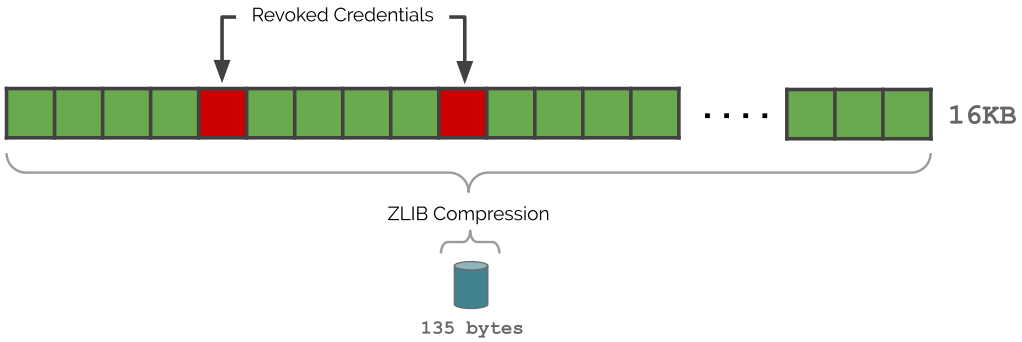


Figura 4.2: Status bitmap. Immagine tratta da [20].

2. Codifica la bitmap in una stringa che viene compressa e inserita nel campo `encodedList` (Listato 4.19).
3. Richiede al Tangle di salvare il modello aggiornato.

```
{
    "@context": [
        "https://www.w3.org/2018/credentials/v1"
    ],
    "type": [
        "VerifiablePresentation",
        "StatusList2021Presentation"
    ],
    "verifiableCredential": {
        "@context": [
            "https://www.w3.org/2018/credentials/v1"
        ],
        "type": [
            "VerifiableCredential",
            "StatusList2021Credential"
        ],
        "issuer": "https://localhost/issuers/565049",
        "issuanceDate": "2021-05-14T15:25:09Z",
        "credentialSubject": {
            "id": "did:ott:KJGJCKX...#list",
            "type": "RevocationList2021",
            "encodedList": "eNoLYGFgYAAAAaUAVQ=="
        },
        ...
        "proof": {
            ...
```

```
            "jws": "eyJhbGciOiJS...LJMQ"
        }
    },
    "proof": {
        "type": "RsaSignatureSuite2018",
        ...
        "jws": "eyJhbGciOi...wr_Q"
    }
}
```

Listato 4.19: Esempio di Status List Presentation.

La classe `StatusBitmap` implementa la logica per gestire lo stato delle credenziali agendo sui bit di una mappa, come mostrato nel Listato 4.20. L'inizializzazione avviene specificando una dimensione, cioè un numero di bit multiplo di 8, in modo tale da rimanere in linea con la dimensione minima di un file salvato su memoria di massa (1 byte). I `_META_BITS` invece sono dei bit riservati a contenere la dimensione corrente della bitmap.

L'operazione di rilascio (`issue`) avviene tramite l'utilizzo della libreria bitarray [10], che fornisce un tipo di struttura dati avente lo stesso funzionamento di un array di booleani. Al rilascio di una credenziale, è sufficiente aggiornare l'indice contenuto nei `_META_BITS`, affinché la dimensione della bitmap sia coerente con il numero di credenziali rilasciate. Con il metodo `frombytes` allora, viene generato un nuovo array di booleani a partire da una variabile costruita mediante la `struct.pack` [41], che crea una rappresentazione in `bytes` del numero che rappresenta l'indice successivo. I bit finali della `_bitmap` saranno sostituiti con il nuovo indice, e la bitmap aggiornata sarà quindi codificata in base64 [38] e poi compressa con zlib [43] (`get_encoded_compressed_string`) per essere ritornata come risultato. Se la bitmap è piena, la stringa sarà nulla, ad indicare l'impossibilità da parte dell'Issuer di rilasciare altre credenziali.

Dato che in Python non esiste un tipo primitivo che implementa i numeri interi senza segno, è necessario affidarsi a librerie terze quali NumPy [24], un modulo molto utilizzato per i calcoli scientifici in ambito Data Science, che offre un tipo di dato a 32 bit senza segno [26]. È semplice dunque ricostruire un numero intero invocando la funzione `frombuffer` [25], in grado di interpretare sequenze di byte in interi senza segno, e di convertirle nel tipo desiderato, come avviene nella `get_next_index`.

Infine, per l'operazione di revoca (`revoke`) è sufficiente porre ad 1 il bit riservato alla credenziale da revocare (`index`), mentre invece, per verificarne lo status (`get_status`) basta esaminarne il valore contenuto.

```
class StatusBitmap(object):
    def __init__(self, size=8):
        self._size = 8 if size < 8 else 8 * round(size / 8)
        self._META_BITS = 4 * 8
        self._bitmap = util.zeros(self._size + self._META_BITS)

```

```
7       def issue(self):
8           next_index = self.get_next_index()
9           if next_index < self.get_size():
10              ni = bitarray()
11              ni.frombytes(struct.pack("I", next_index + 1))
12              self._bitmap[-self._META_BITS :] = ni
13              return self.get_encoded_compressed_string(), next_index
14          elif next_index == self.get_size():
15              return None, next_index
16          else:
17              raise RuntimeError(str(next_index) + " is greater than bitmap
                ↪  length")
18
19      def revoke(self, index):
20          if index < self.get_size() and index < self.get_next_index():
21              if self._bitmap[index] == 1:
22                  return None
23              else:
24                  self._bitmap[index] = 1
25                  return self.get_encoded_compressed_string()
26
27          raise ValueError(str(index) + " out of scope")
28
29      def get_status(self, index):
30          if index < self.get_size() and index < self.get_next_index():
31              return self._bitmap[index]
32          raise ValueError(str(index) + " out of scope")
33
34      def get_next_index(self):
35          next_index = frombuffer(self._bitmap[-self._META_BITS :].tobytes(),
            ↪  uint32)[0]
36          return int(next_index)
37
38      def _get_compressed(self):
39          compressed_bytes = zlib.compress(self._bitmap.tobytes(), 9)
40          return compressed_bytes
41
42      def _get_encoded_compressed(self):
43          compressed_bytes = self._get_compressed()
44          encoded_compressed_bytes = base64.b64encode(compressed_bytes)
45          return encoded_compressed_bytes
46
47      def get_encoded_compressed_string(self):
48          return self._get_encoded_compressed().decode("utf-8")
```

Listato 4.20: Codice per la gestione della bitmap illustrata nella Figura 4.2.

4.2.4 Rilascio di credenziale

Quando un Holder richiede il rilascio di una nuova Verifiable Credential, invia una richiesta che viene gestita dal metodo `handler_issue_credential`, mostrato nel Listato 4.21. I dettagli della richiesta sono contenuti nell'oggetto `IssueCredentialRequestDTO`, che viene ricostruito dal `ModelMapper` partendo dal messaggio in forma di stringa ricevuto dal livello di rete. L'Issuer richiede quindi una resolve per la status list da lui gestita, lasciando che sia il `CredentialStatusService` ad occuparsi dell'aggiornamento della bitmap.

```
async def handle_issue_credential(self, request: str, c: ConnectionContext) ->
↪  str:
    req_dto: IssueCredentialRequestDTO = self._mm.from_json(
        request, IssueCredentialRequestDTO
    )
    sl_pres_dto = await self._did_method_handler.resolve_status_list(
        did=self._status_list_did
    )
    sl_pres_dto = self._credential_status_service.issue(
        status_list_pres_dto=sl_pres_dto
    )
    cred_status_dto = self._credential_status_service.create_credential_status(
        status_list_pres_dto=sl_pres_dto
    )
    cred_dto = self._credential_service.issue_application_credential(
        cred_status_dto=cred_status_dto,
        ...
    )
    _ = await self._did_method_handler.update_status_list(
        sl=sl_pres_dto, ...
    )
    res_dto = ResponseDTO(
        _msg=self._mm.to_json(cred_dto), _code=HTTPStatus.CREATED
    )
    return self._mm.to_json(res_dto)
```

Listato 4.21: Metodo per il rilascio di una nuova Verifiable Credential (classe `Credential_Controller`).

Il metodo `issue` dell'oggetto `CredentialStatusService` (Listato 4.22) fa uso degli oggetti `StatusBitmap`, introdotti nella sottosezione 4.2.3, opportunamente costruiti dalla `get_status_bitmap`. Se la stringa ricevuta è nulla, significa che la bitmap è piena, e quindi viene lanciato un messaggio di errore, altrimenti viene preparato uno `StatusClaimDTO` da inserire nella Status List Credential `sl_cred_dto`, che poi sarà inserita nella Status List Presentation aggiornata.

```
def issue(
    self, status_list_pres_dto: StatusListPresentationDTO
) -> StatusListPresentationDTO:
    bitmap = get_status_bitmap(
        status_list_pres_dto._verifiableCredential._credentialSubject._encoded
        ↪ List
    )
    new_encoded_list, new_index = bitmap.issue()
    if new_encoded_list is None:
        raise StatusListException(f"Status list {status_list_pres_dto._id}
        ↪ full")
    ...
    status_list_cred_dto = status_list_pres_dto._verifiableCredential
    ...
    sl_claim = StatusClaimDTO(
        _encodedList=new_encoded_list,
        ...
    )
    ...
    sl_pres_dto = StatusListPresentationDTO(
        _verifiableCredential=sl_cred_dto,
        ...
    )
    return sl_pres_dto
```

Listato 4.22: Codice per l'aggiornamento della status list (classe `CredentialStatusServi` ↵ `ce`).

L'operazione di rilascio prosegue creando un nuovo `credentialStatus` in un oggetto `CredentialStatusList21DTO`. Listato 4.23 mostra come, sempre attraverso il metodo `cre` ↵ `ate_credential_status` della classe `CredentialStatusService`, la bitmap viene ricostruita affinché l'indice corrente venga recuperato (`get_current_index`) e sia inserito nello status della nuova credenziale.

```
def create_credential_status(self, status_list_pres_dto:
↪ StatusListPresentationDTO) -> CredentialStatusList21DTO:
    bitmap = get_status_bitmap(
        status_list_pres_dto._verifiableCredential._credentialSubject._encoded
        ↪ List
    )
    ...
    new_index = bitmap.get_current_index()
    cred_status_dto = CredentialStatusList21DTO(
        _statusListIndex=new_index,
        ...
```

```
10      )
11      return cred_status_dto
```

Listato 4.23: Codice per la creazione di un nuovo claim da inserire in una credenziale dedicata alla status list (classe `CredentialStatusService`).

Il `CredentialController` quindi, completa il rilascio creando un nuovo `CredentialModel` grazie alla `issue_application_credential` del `CredentialService`, e genera le proof per i modelli della status list. Solo dopo aver completato l'aggiornamento sul Tangle (`updat` ⌋ `e_status_list`) tramite il `_did_metho_handler`, invia all'Holder la risposta contenente la nuova credenziale, che viene salvata nella sua Repository.

4.2.5 Accesso al servizio

Il Verifier espone i propri servizi attraverso i metodi dell'`ApplicationController` registrato sull'endpoint in ascolto. Listato 4.24 mostra che il metodo esegue dapprima una verifica della richiesta (`_verify_request`) e poi estrae il claim dalla credenziale. Tramite un oggetto del tipo `AuthorizationHandler`, il controllo dell'accesso viene valutato in base alle informazioni contenute nel claim estratto, che quindi determina l'autorizzazione di cui gode l'Holder nell'accedere al servizio. Nel caso in cui la credenziale sia sufficiente ad autorizzarlo, il Verifier da accesso alla logica contenuta nell'`ApplicationService` (`read`).

```
1  async def handle_application_read_request(
2      self, request: str, c: ConnectionContext
3  ) -> str:
4      pres_dto, claim_dto = await self._verify_request(request=request, c=c)
5      self._authorization_handler.check_read_authorization(claim=claim_dto)
6      res = self._application_service.read()
7      res_dto = ResponseDTO(_msg=res, _code=HTTPStatus.OK)
8      return self._mm.to_json(res_dto)
```

Listato 4.24: Codice per la gestione di una richiesta di accesso in lettura (classe `Applica` ⌋ `tionController`).

Il processo di verifica avviene seguendo in ordine:

1. Verifica della proof contenuta nella Verifiable Presentation, utilizzando la chiave a crittografia pubblica dell'Holder (Listato 4.25).
2. Verifica della proof contenuta nella Verifiable Credential, utilizzando la chiave pubblica dell'Issuer.
3. Ricerca, nella propria trusted list, dell'Issuer che figura nella credenziale.

4. Verifica della data di rilascio e di scadenza.
5. Verifica della validità dello schema della richiesta ricevuta.
6. Verifica dello status della credenziale (revocata oppure no).

```python
def _verify_presentation_proof(self, p: PresentationDTO, conn:
↪  ConnectionContext):

    pres_dict = self._mm.to_dict(p)
    _ = pres_dict.pop("proof")

    mat = conn._assert_doc._assertionMethod

    try:
        self._verification_strategy.execute(
            payload=pres_dict, proof=p._proof, mat=mat, conn=conn
        )
    except ValueError:
        raise ProofException(f"Presentation {p._id} proof is not valid")
```

Listato 4.25: Codice per la verifica della proof di una Verifiable Presentation (classe `VerificationHandler`).

Dal Listato 4.25 notiamo che il `VerificationHandler` esegue la verifica della proof in maniera astratta. L'oggetto `SignatureVerificationStrategy` utilizzato, viene lanciato per invocare semplicemente la `execute`, che accetta in input il `payload` su cui fare la verifica, la `_proof` che contiene la signature, il materiale crittografico `mat` contenuto nel `Connect` ⌋ `ionContext` e il `ConnectionContext`. Sarà quindi la strategy ad occuparsi dei passaggi di verifica e di segnalare eventuali incongruenze.

```python
def execute(
    self,
    payload: dict,
    proof: JwsProofDTO,
    mat: RsaPublicCryptoMaterialDTO,
    conn: ConnectionContext,
):
    encoded_jws = proof._jws.encode('utf-8')
    encoded_payload = json.dumps(payload).encode("utf-8")
    public_k = mat._publicKeyPem
    jws_protected_header, encoded_signature = encoded_jws.split(b"..")
    signature = decode64(encoded_signature)
    jws_signing_input = b".".join([jws_protected_header, encoded_payload])
```

```
14      key = load_pem_public_key(public_k.encode('utf-8'))
15      key.verify(
16          signature,
17          jws_signing_input,
18          padding.PKCS1v15(),
19          hashes.SHA256()
20      )
```

Listato 4.26: Implementazione della verifica di una signature JWS (classe `JwsVerificationStrategy`).

4.2.6 Revoca di credenziale

Il processo di revoca è relativamente semplice, un Issuer deve infatti assicurarsi che la richiesta fatta dall'Holder sia valida, attraverso la verifica della proof della Verifiable Presentation trasmessa, e poi recuperare la status list, il cui identificativo viene indicato nella Verifiable Credential da revocare.

La `revoke` (Listato 4.27) offerta dal `CredentialStatusService`, si occupa di aggiornare la bitmap che viene ricostruita dalla stringa contenuta nella Status List Presentation. L'operazione procede ponendo ad 1 il bit che corrisponde all'`index` riservato della credenziale inviata dall'Holder. Se l'esito è positivo, la status list viene aggiornata con la nuova stringa derivante dall'operazione conclusa. La richiesta viene quindi completata inviando il nuovo modello di status list al Tangle, e con un messaggio di conferma all'Holder.

```
1   def revoke(
2       self, status_list_pres_dto: StatusListPresentationDTO, index: int,
        ↪  cred_id: str
3   ) -> StatusListPresentationDTO:
4       bitmap = get_status_bitmap(
5           status_list_pres_dto._verifiableCredential._credentialSubject._encoded
            ↪  List
6       )
7       new_encoded_list = bitmap.revoke(index=index)
8       ...
9       sl_pres_dto = StatusListPresentationDTO(
10          _verifiableCredential=sl_cred_dto,
11          ...
12      )
13      return sl_pres_dto
```

Listato 4.27: Codice per la revoca di una credenziale (classe `CredentialStatusService`).

4.2.7 Inoltro di richiesta

Lo scenario dell'inoltro di richiesta vede interagire tre nodi:

- Holder che richiede l'inoltro della richiesta al Relay.
- Relay che riceve la richiesta dall'Holder e provvede ad inoltrare la richiesta ad un Verifier.
- Verifier che riceve la richiesta da parte dell'Holder attraverso il Relay.

Il nodo Relay possiede una doppia identità, una da Holder e l'altra da Verifier. Una viene utilizzata per accedere al secondo Verifier, mentre quell'altra per accettare le connessioni dall'Holder. La logica per l'accesso al servizio offerto dal Relay è la medesima descritta nella sottosezione 4.2.5, in aggiunta però, vi è l'attesa della risposta da parte del Verifier a cui il Relay inoltra la richiesta, che avviene in maniera asincrona e dunque non bloccante.

Il comportamento dell'inoltro è implementato nella classe `RelayHandler`, avente come unica responsabilità quella di preparare la richiesta da trasmettere al Verifier (Listato 4.28). Per realizzare la nuova connessione, il Relay utilizza il `ClientNetworkHandler`, comune ai nodi di tipo Holder, tramite cui specifica gli identificativi del materiale crittografico necessario ad autenticarsi, e l'identificativo di una Verifiable Credential autorizzata ad accedere al servizio.

```
1  class RelayHandler:
2      def __init__(
3          self,
4          auth_crypto_id: str,
5          assert_crypto_id: str,
6          credential_id: str,
7          verifier_id: str,
8          owner: str,
9          client: ClientNetworkHandler,
10         mm: ModelMapper,
11         presentation_service: PresentationService,
12         verifier_service: VerifierService,
13         did_method_handler: DidMethodHandler,
14     ):
15         self._auth_crypto_id = auth_crypto_id
16         self._assert_crypto_id = assert_crypto_id
17         ...
18
19     async def forward_request(self, *args, **kwargs) -> ResponseDTO:
20         try:
21             return await create_application_request(
22                 auth_crypto_id=self._auth_crypto_id,
23                 assert_crypto_id=self._assert_crypto_id,
24                 credential_id=self._credential_id,
25                 verifier_id=self._verifier_id,
```

```
                service_type=VerifierServiceType.WRITE.value[0],
                client=self._client,
                ...
            )
        except Exception:
            raise ForwardException()
```

Listato 4.28: Implementazione della classe `RelayHandler`.

La logica di creazione della richiesta è mostrata nel Listato 4.29. Grazie al `VerifierService`, il Relay legge il DID del Verifier a cui inoltrare la richiesta, e richiede al Tangle il suo DID document, da cui estrae l'indirizzo e la porta verso cui aprire la connessione. Il `PresentationService` crea una Verifiable Presentation, la cui signature viene calcolata utilizzando la chiave identificata da `assert_crypto_id`. L'operazione procede quindi con la trasmissione della richiesta, che mette il Relay in attesa della risposta. Una volta ricevuta allora, questa viene spedita all'Holder che ha richiesto l'inoltro. Dopo che l'invocazione della `forward_request` termina, l'esecuzione riprende nel `ServerNetworkHandler`, come mostrato in Listato 4.10, lasciando il Relay in attesa di nuove connessioni.

Data la natura modulare delle componenti, è possibile lanciare più Relay per lo stesso nodo. Creando infatti più oggetti `ClientNetworkHandler` e `ServerNetworkHandler`, il nodo è in grado inoltrare le richieste utilizzando identità diverse.

```
async def create_application_request(
    auth_crypto_id: str,
    assert_crypto_id: str,
    credential_id: str,
    verifier_id: str,
    service_type: str,
    client: ClientNetworkHandler,
    ...
) -> ResponseDTO:
    try:
        list(map(lambda t: t.value[0],
        ↪  list(VerifierServiceType))).index(service_type)
        verifier_dto = verifier_service.get_verifier(id=verifier_id)
        verifier_doc_dto = await did_method_handler.resolve_document(
            did=verifier_dto._did
        )
        verifier_service: ServiceDTO = list(
            filter(
                lambda t: t._type == service_type,
                verifier_doc_dto._services,
            )
        )[0]
        address, port = re.match(
```

```
            SERVICE_ENDPOINT_REGEX, verifier_service._serviceEndpoint
        ).groups()
        pres_dto = presentation_service.create_application_presentation(
            crypto_id=assert_crypto_id, credential_id=credential_id, ...
        )
        pres_dto_s = mm.to_json(pres_dto)
        res = await client.connect(
            addr=address,
            port=port,
            packet=pres_dto_s,
            crypto_id=auth_crypto_id,
        )
        if res is None:
            raise ConnectionRefusedError("NO RESPONSE FROM VERIFIER")
        res_dto: ResponseDTO = mm.from_json(json_s=res, cls_type=ResponseDTO)
        return res_dto
    except Exception:
        raise
```

Listato 4.29: Creazione di una Verifiable Presentation per l'accesso ad un servizio (modulo `network`).

Capitolo 5

Sperimentazione

In questo capitolo verrà descritta la toolchain utilizzata per la compilazione di un'immagine custom contenente tutte le librerie necessarie all'esecuzione del framework. Seguirà una descrizione del testbed su cui è avvenuta la sperimentazione, l'implementazione di un servizio remoto in grado di gestire l'identità di più dispositivi, e infine quella di un'applicazione dimostrativa, finalizzata a illustrare le funzionalità del framework attraverso un'interfaccia web interattiva.

5.1 Yocto

Il progetto Yocto [66] comprende una toolchain che permette agli sviluppatori di creare distribuzioni Linux custom da installare su dispositivi embedded. L'insieme degli strumenti messi a disposizione definisce un ambiente collaborativo, che permette di creare risorse e configurazioni condivise.

Tra le feature più importanti ricordiamo:

- Compatibilità con qualsiasi architettura: Yocto supporta qualsiasi architettura a condizione che il produttore metta a disposizione i package di supporto dedicati alla board target (BSP), necessari a rendere l'hardware compatibile.
- Flessibilità: la configurazione dell'ambiente di sviluppo può essere utilizzata per architetture multiple.
- Ideale per dispositivi Internet of Things: a differenza di una distribuzione Linux completa, Yocto permette di creare immagini minime che includono solo i package, le librerie e le funzionalità indispensabili. Ad esempio, lo sviluppatore può non includere le componenti di supporto grafico oppure quelle di rete.
- Estensibilità: se l'architettura target non è supportata dalla toolchain standard, è possibile estenderla attraverso parametri di configurazione.
- Modularità: le funzionalità incluse in un'immagine di sistema vengono specificate tramite un sistema a livelli. Ogni livello definisce dei bundle che decrementano la complessità del progetto, raggruppano e isolano le feature comuni.

Sebbene il progetto Yocto vanti di una community estesa e goda di una documentazione estensiva, nel suo utilizzo si va incontro a numerose difficoltà. Ad un primo approccio infatti, la toolchain appare poco intuitiva, motivo per cui la curva di apprendimento si rivela moderatamente ripida. Date le possibilità diverse che la toolchain offre, per completare task simili, il workflow potrebbe risultare confuso e richiedere molto tempo per il debug dei problemi. La più grande limitazione però, è data dall'hardware richiesto per completare la compilazione delle immagini. Infatti, il tempo di build di un'immagine minimale, utile solo ad avviare una board, può arrivare fino ad un'ora, utilizzando una macchina dotata di specifiche fuori dalla portata del consumatore.

5.1.1 Componenti e strumenti di sviluppo

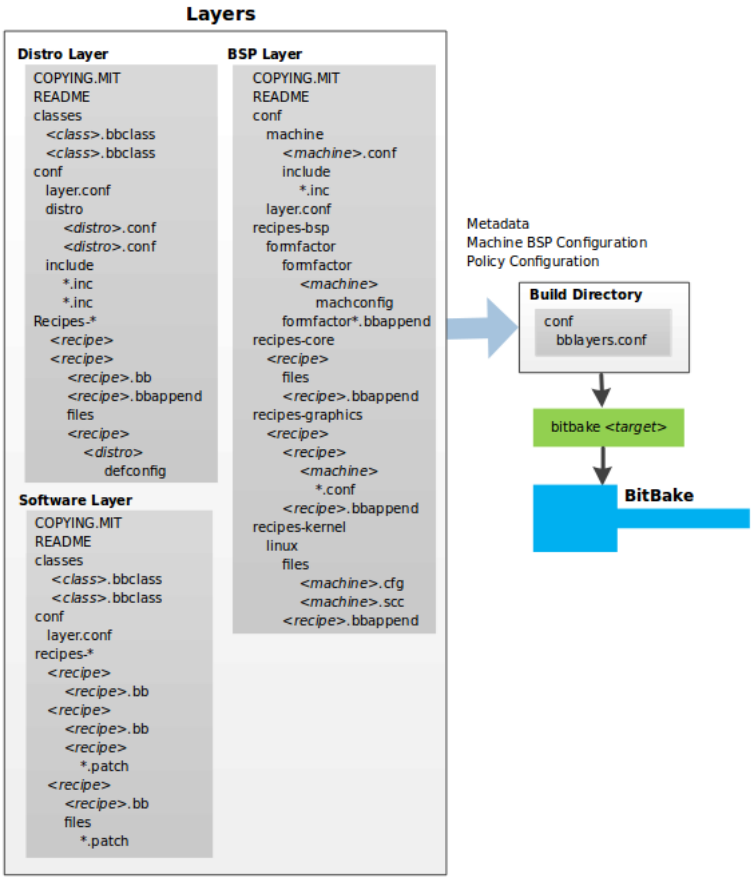


Figura 5.1: Struttura dei livelli indispensabili per una compilazione. Il Distro Layer (policy) e il BSP Layer (package per le board specifiche) vengono forniti dai produttori dell'architettura target. Immagine tratta da [67].

La toolchain del progetto è composta da diverse componenti, ognuna dedicata ad un compito specifico. Nelle seguenti sezioni daremo quindi una breve descrizione degli

strumenti che andranno a realizzare il workflow, e dei parametri di configurazione.

BitBake e OE-Core

BitBake è uno dei componenti core del progetto Yocto ed è utilizzato dall'OpenEmbedded Build System per produrre le immagini. Comprende un motore che effettua il parsing delle ricette e genera i task da eseguire. L'OpenEmbedded-Core è invece un livello che contiene i metadati (ricette, classi e file associati comuni), le definizioni globali e le configurazioni condivise.

Poky

Poky è la distribuzione di riferimento del progetto Yocto. Viene rilasciato in una repository che comprende l'OpenEmbedded Build System (BitBake e OE-Core) e i livelli necessari a compilare distribuzioni minimali.

Recipes

Le recipe (ricette) sono file con estensione `.bb` che contengono le impostazioni e i task, che istruiscono il motore di BitBake sui package da includere nell'immagine da compilare. Una ricetta descrive quindi dove i file sorgente sono contenuti (repository remota o directory locale), quali patch applicare al codice e le opzioni per il processo di compilazione.

Layers

Yocto definisce uno sviluppo secondo un modello strutturato a livelli. I livelli sono repository contenenti set di istruzioni che servono da direttive all'OpenEmbedded Build System per il processo di compilazione. Data la loro natura modulare, i livelli possono essere riutilizzati e condivisi, rendendo i progetti collaborativi. Ogni repository può contenere più livelli separati in diverse directory, ognuna delle quali viene nominata con il prefisso `meta-` per convenzione. Ogni directory (livello) può a sua volta essere ulteriormente suddivisa in altre directory, fino a raggiungere quelle che contengono le recipe. Ogni livello inoltre, comprende una directory `conf` contenente un file `layer.conf` che ne definisce i parametri di configurazione.

layer.conf

Il file `layer.conf` contiene le regole con cui BitBake processa il livello.

```
1 BBPATH .= ":${LAYERDIR}"
2 BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
3             ${LAYERDIR}/recipes-*/*/*.bbappend"
4 BBFILE_COLLECTIONS += "meta-iotsec"
5 BBFILE_PATTERN_meta-iotsec = "^${LAYERDIR}/"
6 BBFILE_PRIORITY_meta-iotsec = "10"
7 LAYERDEPENDS_meta-iotsec = "core"
```

Listato 5.1: File di configurazione del livello `meta-iotsec`.

In riferimento al Listato 5.1:

BBPATH
Aggiunge la directory root del livello al percorso di ricerca di BitBake. Viene utilizzata prima della compilazione per localizzare i file di configurazione.

BBFILE
Definisce il percorso di ricerca delle ricette del livello. Contiene una lista di path da cui il motore di compilazione preleva le ricette da includere nell'immagine.

BBFILE_COLLECTIONS
Definisce un identificativo univoco che viene utilizzato dall'OpenEmbedded Build System per riferirsi al livello.

BBFILE_PRIORITY
Priorità di compilazione per le ricette. Spesso capita di includere livelli aventi ricette con lo stesso nome, il build system allora utilizza questa variabile per valutare con quale priorità devono essere compilate.

LAYERVERSION
Numero di versione del livello.

LAYERDEPENDS
Lista dei livelli su cui il livello dipende.

local.conf

Il file `local.conf` è utilizzato per definire la User Configuration, contenuta nella directory da cui viene fatto partire il processo di build dell'immagine.

```
1 MACHINE ??= 'imx6ul-var-dart'
2 DISTRO ?= 'fslc-framebuffer'
3 IMAGE_INSTALL_append = "\
4         sqlcipher \
5         python3-pytest \
6         python3-bitarray \
7         ...
```

Listato 5.2: File di configurazione della directory di build per la board i.MX6.

In riferimento al Listato 5.2:

`MACHINE`
Architettura target su cui deve essere installata l'immagine.

`DISTRO`
Policy per la compilazione dell'immagine.

`IMAGE_INSTALL_append`
Lista di package aggiuntivi che non sono forniti dall'immagine di base.

bblayers.conf

Il file `bblayers.conf` specifica quali livelli devono inclusi nella compilazione.

```
1 BBPATH = "${TOPDIR}"
2 BBLAYERS = " \
3   ${BSPDIR}/sources/poky/meta \
4   ${BSPDIR}/sources/poky/meta-poky \
5   ${BSPDIR}/sources/meta-variscite-fslc \
6   /home/dev/var-fslc-yocto/sources/meta-iotsec \
7   ...
```

Listato 5.3: Configurazione dei livelli da includere per la compilazione.

In riferimento al Listato 5.3:

`BBLAYERS`
Specifica i path dei livelli che mettono a disposizione le ricette richieste per la compilazione.

5.1.2 Workflow

Il workflow è composto da diversi passaggi preliminari che precedono la compilazione. Definite le configurazioni e la struttura delle directory da cui avviare la toolchain, Yocto invoca le varie componenti, che lavoreranno per produrre i file necessari all'installazione dell'immagine compilata sull'architettura di riferimento. Riassumiamo quindi i passaggi da seguire:

1. Specifica dell'architettura target, delle policy, delle patch e dei dettagli di configurazione all'interno dei file `.conf`.
2. Download dei codici sorgente da parte del build system dalle repository.
3. Estrazione dei sorgenti, applicazione delle patch e delle impostazioni di configurazione.
4. Installazione dei package software in un'area temporanea detta package feed.

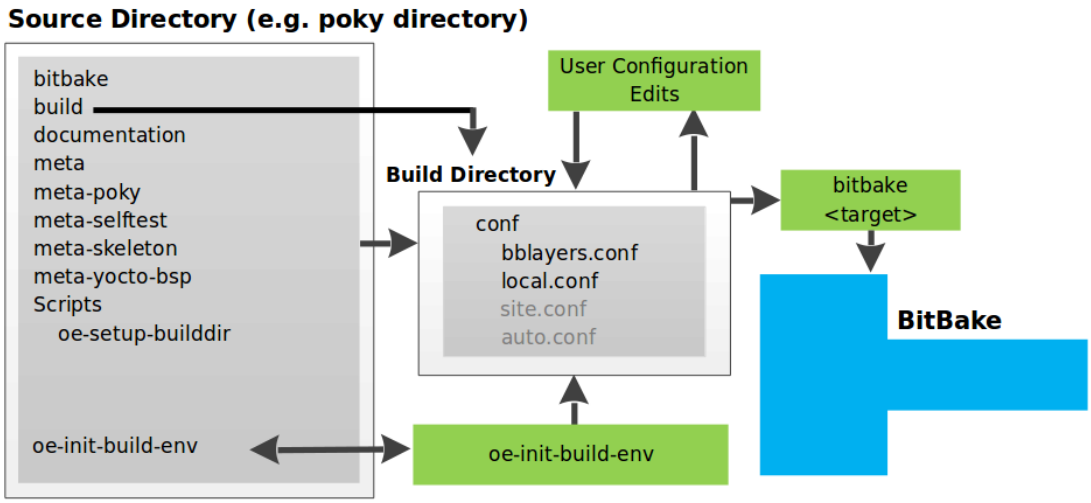


Figura 5.2: Struttura delle directory di Poky. Il file `oe-init-build-env` contiene tutti gli script per Poky e BitBake e permette di generare una cartella preconfigurata per la compilazione. Immagine tratta da [67].

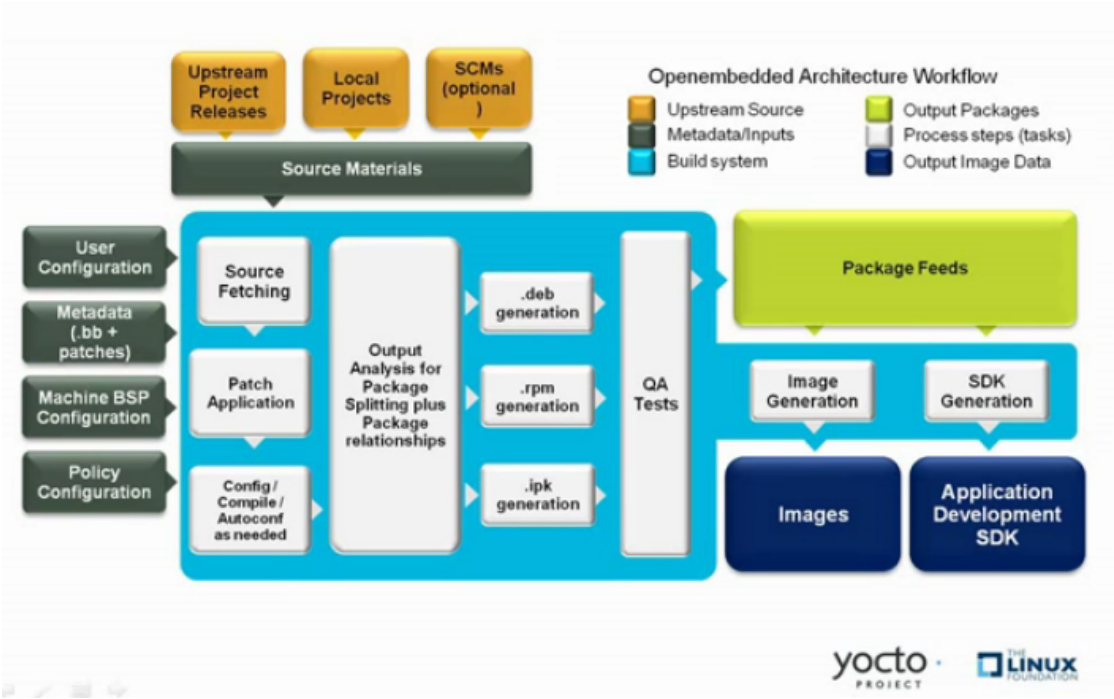


Figura 5.3: Rappresentazione ad alto livello del workflow del sistema di build OpenEmbedded. Immagine tratta da [67].

5. Passaggi di verifica, quali risoluzione delle dipendenze.

6. Generazione del file di immagine, alimentato dai package contenuti nel package feed.

L'output è costituito da file `.dtb` (Device Tree Blob) che descrivono le componenti hardware dell'architettura target, e file immagine contenenti il root file system e il kernel da installare sul dispositivo.

5.2 Testbed e setup

Il banco di prova comprende una board Variscite NXP i.MX DART-6UL e un PC di architettura moderna. Anche se il devkit della scheda è commercializzato con delle immagini già pronte da installare, è stato comunque necessario del lavoro per includere e aggiornare le librerie necessarie all'esecuzione del codice mostrato nel Capitolo 4. Per questa ragione, si è sviluppato un nuovo meta-livello chiamato `meta-iotsec`, che verrà incluso nella toolchain per rendere il framework SSI compatibile con la board target.

La distribuzione realizzata è basata su un'immagine minimale offerta da Poky, `core-image-minimal`, utilizzabile solo per avviare Linux. Una parte del lavoro allora, è stata quella di aggiungere in maniera incrementale tutti i package necessari, mediante la scrittura, la modifica e l'aggiornamento di recipe nuovi e di già esistenti.

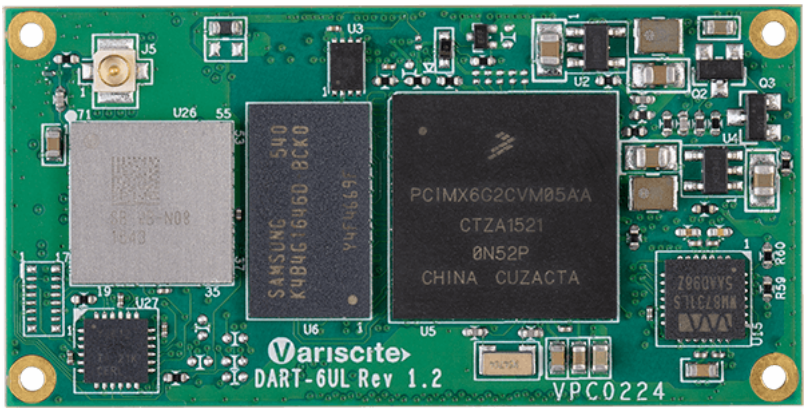


Figura 5.4: NXP i.MX DART-6UL. Immagine tratta da [62].

CPU	Cortex™-A7
CPU core	1 core
CPU clock (max)	900 MHz
RAM	1024 MB DDR3L
NAND	512 MB
eMMC	4 GB
Ethernet	2x 10/100 Mbps

Tabella 5.1: Specifiche della board NXP i.MX DART-6UL.

Uno degli strumenti più utilizzati offerti da Yocto è stato devtool, che permette allo sviluppatore di creare un workspace per lavorare sulle ricette, compilare sorgenti e applicare patch al codice. Nonostante il tool sia di grande utilità, non è sufficiente a garantire una soluzione completa. Il problema infatti, spesso è dovuto alla versione di Yocto supportata dall'architettura target, che non viene aggiornata con i package più recenti e pertanto include ricette obsolete.

```
DESCRIPTION = "Python SQL toolkit and Object Relational Mapper that gives \
application developers the full power and flexibility of SQL"
HOMEPAGE = "http://www.sqlalchemy.org/"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://LICENSE;md5=3359ed561ac16aaa25b6c6eff84df595"
SRC_URI[md5sum] = "236e7d5d31cbd000a60aab39bc2dd07a"
SRC_URI[sha256sum] =
 ↪  "ec1be26cdccd60d180359a527d5980d959a26269a2c7b1b327a1eea0cab37ed8"
PYPI_PACKAGE = "SQLAlchemy"
inherit pypi setuptools3
RDEPENDS_${PN} += " \
    ${PYTHON_PN}-json \
    ${PYTHON_PN}-pickle \
    ${PYTHON_PN}-logging \
    ${PYTHON_PN}-netclient \
    ${PYTHON_PN}-numbers \
    ${PYTHON_PN}-threading \
"
```

Listato 5.4: Ricetta della libreria SQLAlchemy.

Listato 5.4 mostra un esempio di ricetta contenente le direttive comuni ai package che includono moduli Python, sufficienti a preinstallare librerie aggiuntive in un'immagine compilata con Yocto. Tra queste troviamo:

`inherit`
: Permette di ereditare le funzionalità di una classe, intesa come ricetta.

`RDEPENDS_package-name`
: Lista delle dipendenze risolte a runtime.

`SRC_URI`
: Lista degli URI che dei file sorgente di cui fare il fetch, sia in locale che da remoto.

`SRC_URI[*]`
: Checksum da verificare per file non locali.

`LICENSE`
: Tipo di licenza.

`LIC_FILES_CHKSUM`
: Checksum della licenza da verificare.

Una volta definite le ricette da includere nel livello `meta-iotsec` realizzato, il setup viene completato configurando la directory di build tramite le variabili dei file di configurazione descritti nella sottosezione 5.1.1. L'insieme delle direttive specificate dalle ricette e delle configurazioni permette a BitBake di creare una catena di task che determina il processo di compilazione (Figura 5.5).

5.3 Implementazione del servizio

Il concetto di Identity-as-a-Service è inteso come un servizio remoto che espone API REST in grado di servire più Holder, ovvero dei peer installati su dispositivi non aventi le risorse necessarie a gestire la propria identità digitale. La soluzione proposta in questa sezione agirà da back-end per l'applicazione internet che sarà introdotta nella sezione 5.4.

Lo sviluppo è avvenuto utilizzando il framework FastAPI, che permette di realizzare delle API efficienti, robuste e standard-compliant (OpenAPI) [15] in Python. Il framework crea nuovi endpoint analizzando i metodi decorati da un oggetto del tipo `FastAPI` (`app` nel nostro caso), unico per ogni applicazione. Applicando quindi ai metodi un'annotazione del tipo `@app.<method_type>`, all'avvio del server, l'applicazione resta in ascolto sul path specificato nell'annotazione, e processa le richieste invocando il metodo annotato.

Listato 5.5 mostra la definizione di un nuovo endpoint all'indirizzo `http://<addr>:<port>/dids` che serve richieste di tipo POST, e le cui risposte conterranno uno status code uguale a `CREATED`, per segnalare la creazione di una nuova risorsa.

```
1 @app.post("/dids", status_code=HTTPStatus.CREATED)
2 async def add_did(
```

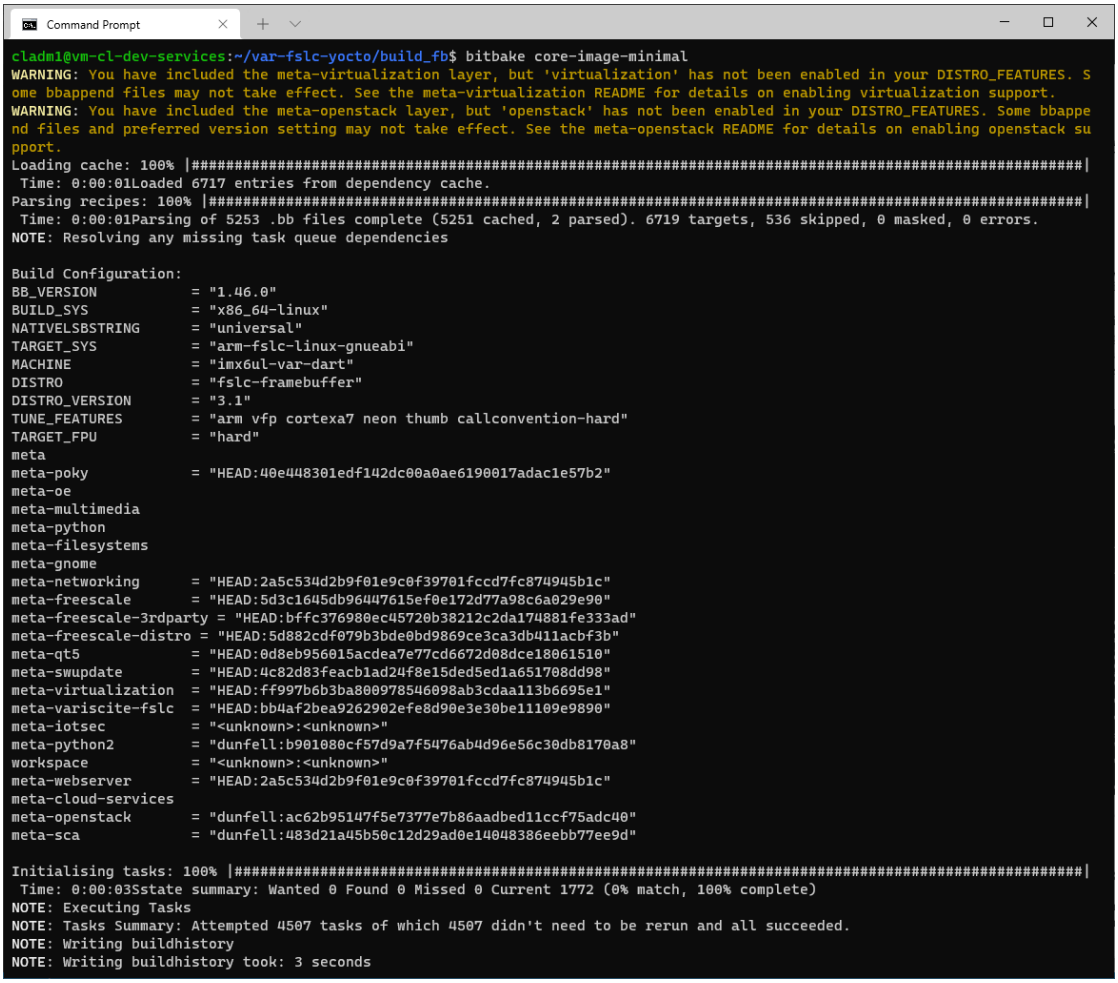


Figura 5.5: Processo di compilazione con Yocto.

```
    req: DidRequestDTO,
    mm: ModelMapper = Depends(mm),
    did_method_handler: DidMethodHandler = Depends(did_method_handler),
    did_service: DidService = Depends(did_service),
    exception_handler: ExceptionHandler = Depends(exception_handler),
):
    try:
        did = await did_method_handler.create_did()
        doc_dto = did_service.create_did(did=did, owner=req.owner, services=[])
        _ = await did_method_handler.create_document(
            doc=doc_dto, owner=req.owner, service_list=[]
        )
        return mm.to_dict(doc_dto)
    except Exception as e:
        exception_handler.raise_http_ex(e)
```

Listato 5.5: Esempio di endpoint per creare un nuovo DID document (file `holder_app.py`).

Nella stringa di definizione del path, FastAPI consente di specificare delle variabili, che possono essere accedute nella definizione del metodo, come mostrato nel Listato 5.6. Questa funzionalità torna particolarmente utile nei casi in cui occorre filtrare i risultati di un'operazione, ad esempio per ottenere le credenziali appartenenti al dispositivo identificato da `owner_id`.

```python
@app.get("/{owner_id}/credentials")
async def get_credential_list(
    owner_id: str,
    mm: ModelMapper = Depends(mm),
    credential_service: CredentialService = Depends(credential_service),
    exception_handler: ExceptionHandler = Depends(exception_handler),
) -> List[dict]:
    try:
        cred_list = credential_service.get_credential_list(owner=owner_id)
        return [mm.to_dict(c) for c in cred_list]
    except Exception as e:
        exception_handler.raise_http_ex(e)
```

Listato 5.6: Esempio di endpoint con variabile (file `holder_app.py`).

Siccome il servizio è pensato per riutilizzare le classi definite nel Capitolo 3, è necessaria una fase di inizializzazione prima dell'avvio del server. FastAPI permette di registrare delle funzioni che saranno invocate al verificarsi di eventi, per cui è sufficiente definire in un metodo decorato con l'annotazione `@app.on_event("startup")`, il codice da eseguire all'avvio del server.

Affinché tutti gli endpoint possano richiedere le istanze degli oggetti del framework, necessari per offrire le loro funzionalità, occorre salvare lo stato dell'applicazione in una mappa contenente tutti gli oggetti inizializzati all'avvio. `app`, cioè l'oggetto che rappresenta l'applicazione, contiene una variabile `state` definita per utilizzi arbitrari. Al suo interno saranno quindi inserite le istanze dei Service e degli Handler necessari a realizzare i casi d'uso introdotti nel Capitolo 4.

```python
@app.on_event("startup")
async def boot():
    mm, jws_verification_strategy, verification_handler, ... = await init(
        tangle_address=tangle_address,
        tangle_port=tangle_port,
        db_name=HOLDER_APP_DB_NAME,
```

```
        secret=secret,
    ) # inizializzazione Repository, Handler, Service
    ...
    app.state = {
        "mm": mm,
        "did_service": did_service,
        "credential_service": credential_service,
        "presentation_service": presentation_service,
        "issuer_service": issuer_service,
        "verifier_service": verifier_service,
        "client_handler": client_handler,
        "did_method_handler": did_method_handler,
        "exception_handler": exception_handler,
        "credential_status_service": credential_status_service,
    }
```

Listato 5.7: Codice di inizializzazione del servizio (file `holder_app.py`).

L'istruzione `Depends` permette di definire una dipendenza in maniera esplicita. Nel nostro caso, come mostrato nel Listato 5.5, i metodi che definiscono gli endpoint necessitano degli oggetti contenuti nello stato dell'applicazione. La risoluzione delle dipendenze avviene quindi mediante delle callback (Listato 5.8), che estraggono le istanze create all'avvio dalla variabile `state` dell'oggetto `app` (Listato 5.7). Per servirsi di questo meccanismo è sufficiente inserire le istruzioni di `Depends` nella signature del metodo e specificare la callback da invocare. All'avvio del server allora, FastAPI inizializza ogni endpoint fornendo le dipendenze richieste.

Un approccio simile può sembrare poco pratico in termini di duplicazione del codice, tuttavia la scelta di rendere esplicite le dipendenze per ogni endpoint è molto utile in fase di testing, come verrà mostrato nel Capitolo 6.

```
async def did_service():
    yield app.state["did_service"]

async def credential_service():
    yield app.state["credential_service"]
```

Listato 5.8: Esempi di callback che recuperano le istanze degli oggetti salvati nello stato dell'applicazione, per risolvere le dipendenze richieste tramite le istruzioni di `Depends` (file `holder_app.py`).

5.3.1 Ricezione delle richieste e gestione delle eccezioni

Nelle signature dei metodi su cui vengono registrati gli endpoint, è possibile specificare un oggetto che descrive i dati della richiesta ricevuta.

La libreria che mette a disposizione i meccanismi di conversione e di validazione dei dati è Pydantic [34]. In pratica, le richieste della rete vengono convertite in oggetti definiti dall'utente, cioè classi che estendono il tipo `BaseModel`. In questo modo, gli oggetti vengono arricchiti con le funzionalità del framework e facilitano le operazioni sui dati, secondo la stessa logica delle classi DTO. In fase di ricezione quindi, l'endpoint analizza il body della richiesta, e valida il suo contenuto facendo un match con i campi dell'oggetto `BaseModel`: in caso positivo prosegue con l'azione richiesta fino a generare un risultato del tipo `Response`, in caso negativo risponde con un messaggio di errore.

```
class CreateIssueCredentialRequestDTO(BaseModel):

    subjectDid: str
    authCryptoId: str
    issuerId: str
    categoryType: str
    actionType: str
    owner: str
```

Listato 5.9: Esempio di richiesta per richiedere il rilascio di una nuova credenziale. Il body della richiesta POST viene automaticamente convertito nel tipo di oggetto che estende `BaseModel`.

Per incrementare la robustezza del codice e garantire una risposta al client che richiede il servizio, è necessario gestire tutte le condizioni di errore catturando ogni eccezione. La logica di gestione è definita nella classe `ExceptionHandler`, in cui è definito il metodo `raise_http_ex`, che si occupa di analizzare il tipo di eccezione sollevata e di preparare una risposta contenente un codice di status opportuno. La classe viene inizializzata con una mappa, che registra per ogni condizione di errore un codice di stato HTTP compatibile. Il mapping ha come finalità quella di rilanciare un secondo tipo di eccezione, `HTTPException`, in modo da permettere a FastAPI di riconoscere l'occorrenza di un'eccezione. L'applicazione quindi, interrompe l'esecuzione del codice e invia una risposta contenente i dettagli dell'errore estratti dal `_dispatcher` con cui l'`ExceptionHandler` è stato costruito.

```
def __init__(self, mm: ModelMapper, name: str = ""):
    self._dispatcher = {
        ValueError: ResponseDTO(
            _msg="", _code=HTTPStatus.UNPROCESSABLE_ENTITY
        ),
        ModelNotFoundException: ResponseDTO(
            _msg="", _code=HTTPStatus.NOT_FOUND
        ),
        ...
def raise_http_ex(self, ex: Exception):
    res = ResponseDTO(_msg="", _code=StatusCode.INTERNAL_SERVER_ERROR.value)
```

```
12      try:
13          res: ResponseDTO = self._dispatcher[ex.__class__]
14      except Exception:
15          traceback.print_exc()
16      raise HTTPException(status_code=res._code, detail=res._msg)
```

Listato 5.10: Metodo della classe `ExceptionHandler` per la cattura delle eccezioni e la preparazione del messaggio di errore. `self._dispatcher` è un oggetto di tipo `dict` che contiene le eccezioni derivate.

5.3.2 API

Gli endpoint sviluppati per il servizio introdotto nella sezione 5.3 offrono tutte le funzionalità necessarie ad un dispositivo di tipo Holder per gestire la propria identità digitale. Le seguenti API sono però state implementate per servire più dispositivi, quelli che nell'ecosistema Self-Sovereign possiamo definire come nodi edge, ovvero quei dispositivi che hanno bisogno dell'Identity-as-a-Service per interagire con gli altri peer.

POST /dids
Crea un nuovo DID, il materiale crittografico e fa richiesta al Tangle per salvare il relativo DID document.

GET /owner_id/keys
Ritorna la lista completa delle chiavi RSA pubbliche del dispositivo identificato da owner_id.

GET /owner_id/dids
Ritorna la lista completa dei DID document del dispositivo identificato da owner_id.

GET /issuers
Ritorna la lista completa degli Issuer noti, ovvero la trusted list.

GET /verifiers
Ritorna la lista completa dei Verifier noti.

GET /verifiers/services
Ritorna la lista di tutti i servizi offerti da tutti i Verifier noti.

GET /owner_id/credentials
Ritorna la lista delle credenziali rilasciate al dispositivo identificato da owner_id.

POST /presentations/app
Invia una richiesta di accesso utilizzando una Verifiable Credential. Nel body della richiesta, l'Holder inserisce i seguenti campi:

- identificativo della chiave RSA da utilizzare nel processo di autenticazione;

- identificativo della chiave RSA con cui firmare la presentazione;
- identificativo della credenziale da inserire nella presentazione;
- identificativo del Verifier a cui inviare la richiesta;
- tipo di servizio richiesto;
- proprio identificativo di owner.

Ottenute le informazioni del Verifier verso cui aprire la nuova connessione, viene fatta una richiesta al Tangle per ottenere il suo DID document (noto). Viene preparata una Verifiable Presentation per inserire la credenziale e poi viene inviata al Verifier. Al termine della connessione con il peer, l'Holder riceve il risultato del Verifier in formato JSON.

POST /presentations/issue

Invia una richiesta di rilascio credenziale ad un Issuer. Nel body della richiesta, l'Holder inserisce:

- il DID da inserire nella credenziale;
- identificativo della chiave RSA da utilizzare nel processo di autenticazione;
- identificativo dell'Issuer a cui inviare la richiesta;
- categoria del dispositivo;
- tipo di azione;
- proprio identificativo di owner.

A partire dall'identificativo dell'Issuer, il servizio legge il DID document dell'Issuer dal Tangle. Estrae l'indirizzo dell'endpoint su cui l'Issuer riceve le richieste di rilascio, e preparata una Verifiable Presentation con tutte le informazioni necessarie. Alla ricezione della nuova credenziale, il servizio salva in locale la Verifiable Credential e la invia come messaggio di risposta all'Holder.

POST /presentations/revoke

Invia una richiesta di revoca ad un Issuer. La richiesta deve includere:

- identificativo della credenziale da revocare;
- identificativo della chiave RSA da utilizzare nel processo di autenticazione;
- identificativo della chiave RSA per la firma della signature della presentazione;
- proprio identificativo di owner.

Letto il DID document dell'Issuer dal Tangle, viene selezionato l'endpoint su cui l'Issuer riceve le richieste di revoca. Dopo aver ottenuto il risultato dell'operazione, il servizio provvede ad inoltrarla all'Holder servito.

GET /owner_id/credentials/credential_id/status
Legge la Verifiable Credential identificata da credential_id, appartenente al dispositivo identificato da owner_id e richiede, all'Issuer che si è occupato di rilasciarla, lo status corrente.

GET /issuers/services/types
Ritorna la lista di tutti i tipi servizio erogati dai nodi Issuer, di cui il dispositivo si fida, quindi nel nostro caso rilascio e revoca di credenziale.

GET /verifiers/services/types
Ritorna la lista di tutti i tipi di servizio erogati dai Verifier noti.

GET /credentials/claims/actions
Ritorna la lista di tutti i tipi di azione che è possibile richiedere in fase di rilascio di una nuova Verifiable Credential.

GET /credentials/claims/categories
Ritorna la lista delle categorie di dispositivo IoT che possiedono Verifiable Credential.

GET /issuers/issuer_id:path/services/service_id
Ritorna lo stato (attivo oppure no) del servizio identificato da service_id[1], erogato dall'Issuer identificato da issuer_id.

GET /issuers/issuer_id:path/services
Ritorna lo stato di tutti i servizi erogati dall'Issuer identificato da issuer_id.

GET /verifiers/verifier_id:path/services/service_id
Ritorna lo stato del servizio identificato da service_id, erogato dal Verifier identificato da verifier_id.

GET /verifiers/verifier_id:path/services
Ritorna lo stato di tutti i servizi erogati dal Verifier identificato da verifier_id.

DELETE /owner_id/credentials
Elimina tutte le credenziali appartenenti al dispositivo identificato da owner_id.

DELETE /credentials
Metodo di utilità. Elimina tutte le credenziali salvate dal servizio.

DELETE /owner_id/dids
Invia una richiesta al Tangle per eliminare i DID document appartenenti al dispositivo identificato da owner_id e il suo materiale crittografico.

DELETE /dids
Metodo d'utilità. Elimina tutti i DID sul Tangle e il materiale crittografico salvato sul servizio.

[1]:path permette di inviare stringhe in formato URI senza che alcuni simboli vengano convertiti in fase di parsing.

DELETE /owner_id/status-lists
Metodo d'utilità. Elimina tutte le status list appartenenti all'Issuer identificato da owner_id, facendo una richiesta al Tangle.

DELETE /status-lists
Metodo d'utilità. Elimina tutte le status list di tutti gli Issuer, facendo una richiesta al Tangle.

5.3.3 Esempio di richiesta al servizio

Mostriamo quindi un esempio entrando nei dettagli implementativi, a partire dall'invio della richiesta fino alla risposta da parte del servizio. Listato 5.12 mostra l'implementazione di un metodo che registra un endpoint all'indirizzo `http://<address>:<port>/presentations/app`, utilizzato per creare una Verifiable Presentation e accedere al servizio di un Verifier.

La registrazione dell'endpoint avviene mediante l'oggetto `app`, istanza della classe `FastAPI`, che effettua un binding del metodo `add_app_presentation` all'indirizzo mostrato sopra. Quando il server riceve una richiesta di tipo POST, esegue il parsing del body del messaggio e lo converte in un oggetto `CreateApplicationRequestDTO`, che non è nient'altro che una semplice dataclass derivata da un `BaseModel` (Listato 5.11).

```python
class CreateApplicationRequestDTO(BaseModel):

    credentialId: str
    authCryptoId: str
    assertCryptoId: str
    verifierId: str
    serviceType: str
    owner: str
```

Listato 5.11: Classe contenente i campi necessari per una richiesta di accesso ad un Verifier.

Dopo aver validato la richiesta, il servizio procede a risolvere le dipendenze necessarie invocando le `Depends` specificate nella signature del metodo. Le callback registrate in fase d'avvio allora, leggono le istanze degli oggetti contenenti la logica applicativa dallo stato dell'applicazione. Dal messaggio di richiesta vengono estratti tutti i campi contenuti nell'oggetto `req` in apposite variabili, che servono per verificare l'esistenza del tipo di servizio richiesto e l'esistenza dei modelli richiesti. In caso negativo infatti, viene lanciata un'eccezione ad indicare che la richiesta non è valida, e l'esecuzione viene interrotta. Il `verifier_id` viene utilizzato dal `VerifierService` per leggere la lista dei Verifier noti, da cui recuperare il DID necessario alla resolve sul Tangle, che avviene mediante il `did_method_handler`. Letto il DID document, viene analizzata la proprietà `services` che contiene la lista di tutti i servizi erogati dal Verifier, e i relativi endpoint verso cui aprire la nuova connessione. L'oggetto `presentation_service` implementa la logica per preparare la Verifiable Presentation, in cui verrà inserita la credenziale identificata da `credential_id`, e

la proof contenente la signature creata utilizzando la chiave RSA privata identificata da `assert_crypto_id`.

La nuova connessione viene quindi aperta grazie all'oggetto `client_handler`, che si occupa di trasmettere la Verifiable Presentation, opportunamente serializzata tramite il `mm` (`ModelMapper`). Quest'ultimo viene nuovamente invocato per convertire la risposta ricevuta (stringa JSON) in un oggetto DTO (`response`) di convenienza, da cui vengono estratti i dettagli del messaggio e lo status code dell'operazione, che sarà utilizzato per costruire la risposta per l'Holder.

Notiamo come tutto il codice è incluso in un costrutto `try-except`, che catturerà tutte le eccezioni lanciate in caso di errore. In questo caso, l'`ExceptionHandler` provvede ad analizzare il tipo di eccezione e a rilanciare una seconda eccezione del tipo `HTTPException`. FastAPI riconoscerà quindi la condizione di errore e risponderà con un messaggio di risposta compatibile con l'architettura REST.

```
@app.post("/presentations/app", status_code=HTTPStatus.OK)
async def add_app_presentation(
    req: CreateApplicationRequestDTO,
    response: Response,
    mm: ModelMapper = Depends(mm),
    verifier_service: VerifierService = Depends(verifier_service),
    presentation_service: PresentationService = Depends(presentation_service),
    did_method_handler: DidMethodHandler = Depends(did_method_handler),
    client_handler: ClientNetworkHandler = Depends(client_handler),
    exception_handler: ExceptionHandler = Depends(exception_handler),
) -> str:

    try:
        auth_crypto_id = req.authCryptoId
        assert_crypto_id = req.assertCryptoId
        credential_id = req.credentialId
        verifier_id = req.verifierId
        service_type = req.serviceType
        owner = req.owner

        list(map(lambda t: t.value[0],
        ↪  list(VerifierServiceType))).index(service_type)

        verifier_dto = verifier_service.get_verifier(id=verifier_id)

        verifier_doc_dto = await did_method_handler.resolve_document(
            did=verifier_dto._did
        )

        verifier_service: ServiceDTO = list(
            filter(
                lambda t: t._type == service_type,
                verifier_doc_dto._services,
```

```
33              )
34          )[0]
35
36          address, port = re.match(
37              SERVICE_ENDPOINT_REGEX, verifier_service._serviceEndpoint
38          ).groups()
39
40          pres_dto = presentation_service.create_application_presentation(
41              crypto_id=assert_crypto_id, credential_id=credential_id,
                ↪  owner=owner
42          )
43          pres_dto_s = mm.to_json(pres_dto)
44
45          res = await client_handler.connect(
46              addr=address,
47              port=port,
48              packet=pres_dto_s,
49              crypto_id=auth_crypto_id,
50          )
51
52          if res is None:
53              raise InvalidResponse()
54
55          res_dto: ResponseDTO = mm.from_json(json_s=res, cls_type=ResponseDTO)
56          response.status_code = res_dto._code
57
58          return res_dto._msg
59      except Exception as e:
60          exception_handler.raise_http_ex(e)
```

Listato 5.12: Codice completo dell'endpoint responsabile di inviare una nuova richiesta al Verifier utilizzando una presentazione (file `holder_app.py`).

5.4 Applicazione dimostrativa

L'applicazione che verrà introdotta in questa sezione è stata sviluppata a fini dimostrativi, per i casi d'uso descritti nel Capitolo 4. Il front-end realizzato, che si serve del servizio descritto nella sezione 5.3, offre un'interfaccia user-friendly per interagire con i peer che fanno parte dell'ecosistema Self-Sovereign, dalla prospettiva di un Holder. Lo scopo è quello di poter apprezzare visivamente il funzionamento del framework, che agisce da back-end in un'architettura sperimentale. Nelle sezioni a seguire ne verrà quindi descritta la composizione, illustrando nel dettaglio le tecnologie adottate e come vengono applicate.

5.4.1 Angular e struttura dell'applicazione

Angular [3] è un framework che permette di sviluppare single-page application in TypeScript, un linguaggio fortemente tipizzato che estende il JavaScript. Possiamo definire una single-page application come un'applicazione internet che carica una pagina singola, le cui risorse vengono caricate dal server in maniera dinamica, solo quando richieste. Questo tipo di funzionamento ha come obiettivo quello di fornire un'esperienza più fluida rispetto alle applicazioni web classiche, che invece richiedono il caricamento di intere pagine.

Le applicazioni sviluppate con Angular sono composte da blocchi di Component. Un Component racchiude una classe TypeScript decorata con l'annotazione `@Component()`, un template HTML e un selettore CSS. Il selettore viene inserito all'interno del codice HTML, cioè nei file che descrivono la composizione grafica delle pagine, per creare un'istanza del Component, mentre il template istruisce Angular su come visualizzarne il contenuto. Le regole di stile degli elementi per il template, vengono invece specificate in documenti `.css` distinti. I Component e i relativi template quindi, definiscono le view che compongono le pagine, e che possono essere create, modificate e distrutte singolarmente. Durante la navigazione allora, ogni pagina viene inizializzata creando istanze di Component, che all'uscita verranno distrutte. Questi tipi di evento implementano i lifecycle hook dei Component, e ne descrivono il ciclo di vita. Come viene mostrato nel Listato 5.13, è possibile definire dei metodi nella classe contenente la logica del Component, per reagire a questi eventi. `OnInit`, `AfterViewInit`, `OnDestroy` sono i lifecycle hook che rispondono agli eventi di creazione, di caricamento e di distruzione del Component. Essi tornano estremamente utili per inizializzare variabili, arricchire gli elementi della DOM (Document Object Model) con i dati caricati dal server, oppure liberare le risorse richieste da task eseguiti in background, ed evitare problemi di memory leak.

```
1  @Component({
2    selector: 'app-did-selector',
3    templateUrl: './did-selector.component.html',
4    styleUrls: ['./did-selector.component.css']
5  })
6  export class DidSelectorComponent implements OnInit, OnDestroy, AfterViewInit {
7    @Output() select
8    selectedDid: string | undefined
9    constructor(...) {...}
10   ngOnInit(): void {...}
11   ngAfterViewInit(): void {...}
12   ngOnDestroy(): void {...}
13   selectDid(): void { this.select.emit(this.selectedDid) }
```

Listato 5.13: Esempio di Component Angular. `app-did-selector` è il selettore CSS che inserito in un template HTML crea un'istanza del Component `DidSelectorComponent`, in questo caso `<app-did-selector></app-did-selector>`.

I Component sono raggruppati in classi modulo definite NgModule. I moduli definiscono un insieme di feature correlate, che solitamente fanno parte di un dominio applicativo comune. Alla radice della struttura dell'applicazione vi è il root module, che importa tutti gli altri moduli, definisce l'entry point dell'applicazione ed esegue il bootstrap di tutte le componenti.

Ogni `@NgModule` (Listato 5.14) viene costruito specificando degli array. I Component che sono parte del modulo vengono dichiarati all'interno di `declarations`, mentre invece `imports` permette di includere *altri* moduli, e di ottenere l'accesso ai Component esportati dall'array `exports`. L'approccio potrebbe risultare poco intuitivo, ma rappresenta un modo per ridurre la coesione tra i Component e allo stesso tempo favorirne il riutilizzo, definendo in maniera esplicita le dipendenze che intercorrono.

In Angular esiste un altro tipo di oggetto, chiamato Service, aventi un ciclo di vita indipendente dai Component. Le istanze dei Service vengono fornite mediante un meccanismo di risoluzione definito dependency injection, di cui il modulo si fa carico attraverso l'array `providers`, che ne dichiara la responsabilità.

Infine l'array `bootstrap` specifica il componente da creare all'avvio dell'applicazione, e viene quindi definito solo per il root module.

```
@NgModule({
  declarations: [
    CredentialListComponent,
    CredentialRevokeComponent,
    ...
  ],
  imports: [
    IssuerModule,
    ...
  ],
  exports: [],
  providers: []
})
export class CredentialModule { }
```

Listato 5.14: Modulo `CredentialModule` che include tutte le funzionalità relative alla gestione delle credenziali.

5.4.2 Navigazione

Nelle single-page application sviluppate in Angular, la navigazione è implementata nell'oggetto `Router`, uno speciale Service utilizzato dai Component per accedere alle informazioni relative alla navigazione. Il `Router` viene costruito leggendo le route registrate nel modulo `AppRoutingModule`, che contiene un array dei path, ovvero gli indirizzi delle pagine verso cui l'utente può navigare. Per ogni URL visitato, il `Router` interpreta l'indirizzo della

Route	Component
`/device/:owner`	`HomeComponent` `WelcomeComponent`

`/device/:owner/dids`	`DidContComponent`
	`DidListComponent`
	`DidCreateComponent`

`/device/:owner/keys`	`KeyListComponent`

`/device/:owner/credentials`	`CredentialContComponent`
	`CredentialListComponent`
	`DeviceSelectorComponent`
	`ActionSelectorComponent`
	`IssuerListComponent`
	`DidSelectorComponent`

`/device/:owner/credentials/:id/use`	`CredentialUseComponent`
	`DidSelectorComponent`
	`ServiceSelectorComponent`
	`VerifierListComponent`

`/device/:owner/credentials/:id/revoke`	`CredentialRevokeComponent` `DidSelectorComponent`
`/device/:owner/issuers`	`IssuerContComponent` `IssuerListComponent`
`/device/:owner/verifiers`	`VerifierContComponent` `VerifierListComponent`
`/device/:owner/status-lists`	`StatusListPresentationComponent`
`/device/:owner/settings`	`SettingsComponent` `ConfirmDialogComponent`

Tabella 5.2: Lista delle route con componenti utilizzati. Le pagine che includono path comuni creano Component di entrambe le route, ad esempio `/device/:owner/dids` include gli stessi Component della pagina `/device/:owner`.

pagina, e di conseguenza crea un'istanza dei Component registrati su quello specifico path. Per definire un nuovo percorso di routing occorre quindi specificare:

- `path`: indirizzo interpretato dal `Router`.
- `component`: il Component da creare alla navigazione.
- `children`: Component figli, cioè quelli contenuti nei Component creati mediante il `RouterOutlet`.
- `resolve`: data provider attivato prima della navigazione.

La struttura dell'array `routes` determina quindi i percorsi di navigazione dell'applicazione, che vengono registrati con il metodo `forRoot` della classe `RouterModule`. Quest'ultima si occupa di creare l'istanza del servizio `Router`, che verrà *iniettato* nei Component che la richiedono.

Se analizziamo l'esempio nel Listato 5.15, possiamo notare che per tutti i percorsi che derivano da `/device/:owner`[2], oltre a creare un'istanza di `HomeComponent`, vengono create quelle dei Component `children`, inclusi sotto lo stesso percorso.

```
const routes: Routes = [
{
  path: "device",
  children: [
    {
      path: ":owner",
      component: HomeComponent,
      resolve: {
        owner: OwnerResolverService
      },
      children: [
        {
          path: "dids",
          component: DidContComponent
        },
        ...
{ path: "", redirectTo: "/device/HOLDER", pathMatch: "full" },
{ path: "**", component: PageNotFoundComponent }
];
@NgModule({
imports: [RouterModule.forRoot(routes, { enableTracing: false,
↪  paramsInheritanceStrategy: "always" })],
exports: [RouterModule]
})
export class AppRoutingModule { }
```

Listato 5.15: Modulo `AppRoutingModule`. `path` consente di registrare percorsi utilizzando wildcard come `**` che vengono attivati se non c'è nessun percorso definito per una specifica route.

Angular utilizza il `RouterOutlet` per realizzare la navigazione all'*interno* dei Component. `HomeComponent` definisce nel proprio template il selettore `<router-outlet></router-outlet>` che serve da placeholder dinamico. Se la navigazione avviene tramite i `routerLink` (Listato 5.16), il servizio di routing si attiva e analizza l'URL. Quando il path coicide con

[2] `:owner` è inteso come identificativo univoco del dispositivo.

quello definito per un Component figlio, ad esempio `/device/:owner/dids`, il `<router-outlet></router-outlet>` viene sostituito con un'istanza del `DidContComponent`, registrato su quella route.

```
<!-- link verso la pagina /device/:owner/credentials -->
<a mat-list-item routerLink="credentials" ...>
    <mat-icon matListIcon>description</mat-icon>
    <span matLine>Credentials</span>
</a>
    ...
<mat-sidenav-content>
    <router-outlet></router-outlet>
</mat-sidenav-content>
```

Listato 5.16: Template di `HomeComponent`, selettori per la navigazione con il `RouterOutlet`.

Spesso i Component hanno bisogno di validare le variabili contenute nell'URL, oppure semplicemente di verificare che non siano nulle. Per non duplicare il codice dedicato al controllo di questi dati, che altrimenti andrebbe specificato in ogni Component caricato a quell'indirizzo, Angular mette a disposizione il servizio di `Resolve`. Listato 5.17 mostra l'implementazione di un Resolver che ha il compito di verificare la presenza dell'identificativo del `device` richiesto, e lo fa semplicemente analizzando la `route`, che corrisponde al path di navigazione. In caso negativo quindi, viene attivata una navigazione automatica verso la pagina di root, poichè la condizione specificata non è soddisfatta. Questo controllo appena descritto è fondamentale, dato che le pagine dedicate alla visualizzazione dei contenuti di un determinato dispositivo, richiedono dei parametri da utilizzare per la richiesta delle risorse al server, con cui completare le pagine. Senza queste informazioni, la richiesta non può avvenire e di conseguenza neanche il caricamento della pagina. In Figura 5.6 vi è un esempio, i Component della pagina `/device/:owner/dids` richiedono il parametro `:owner` affinchè possano leggere dal server i DID document che appartengono a quel dispositivo.

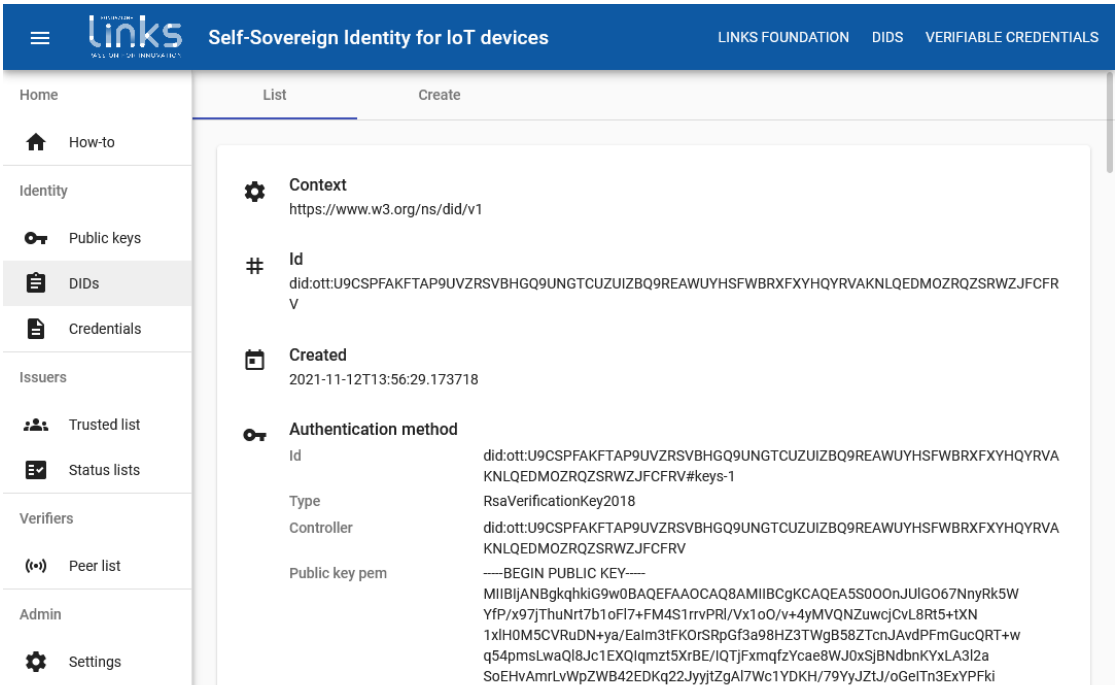
```
@Injectable({
  providedIn: 'root'
})
export class OwnerResolverService implements Resolve<string> {
  constructor(private router: Router) { }

  resolve(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): string | Observable<string> | Promise<string> {
    const owner = route.paramMap.get("owner")

    if (!owner) {
```

```
14        this.router.navigate(["/"])
15        return EMPTY
16      }
17
18      return of(owner)
19    }
20  }
```

Listato 5.17: Esempio di `Resolver` attivato per verificare che l'URL contiene le informazioni necessarie per completare la navigazione.



(a)

(b)

Figura 5.6: Pagina `/device/:owner/dids`.

Route	Funzione
`/device/:owner`	Carica il menu di navigazione e la toolbar principale
`/device/:owner/dids`	Mostra la lista dei DID document posseduti dal device `id`
`/device/:owner/keys`	Mostra la lista delle chiavi RSA pubbliche contenute nei DID document posseduti dal dispositivo `id`
`/device/:owner/credentials`	Mostra la lista delle credenziali rilasciate al dispositivo `id`
`/device/:owner/credentials/:id/use`	Consente di accedere al servizio di un Verifier utilizzando la credenziale `id`
`/device/:owner/credentials/:id/revoke`	Consente di inviare una richiesta di revoca per la credenziale `id`
`/device/:owner/issuers`	Mostra la lista degli Issuer di cui il dispositivo `id` si fida
`/device/:owner/verifiers`	Mostra la lista di Verifier conosciuti dal dispositivo `id`
`/device/:owner/status-lists`	Mostra la lista delle status list
`/device/:owner/settings`	Espone metodi d'utilità

Tabella 5.3: Lista delle route con relative funzionalità.

5.4.3 Presentazione dei contenuti

La modularità che caratterizzano le applicazioni in Angular spinge lo sviluppatore verso il riutilizzo dei Component. Tipicamente la suddivisione si articola in Component detti smart e altri detti dummy. I primi hanno il compito di interfacciarsi con i Service, e si occupano di effettuare il fetch dei dati; i secondi invece ottengono i dati dai Component

smart, e li preparano per la visualizzazione sulla pagina. Tra le due tipologie si crea quindi una relazione parent-child, in cui il Component dummy dipende da quello smart.

La visualizzazione dei dati ottenuti avviene attraverso i meccanismi di binding. Consideriamo il `DidSelectorComponent`, che rappresenta un esempio di smart Component, in quanto, servendosi del `DidService` legge dal server la lista dei DID che appartengono al dispositivo, e gestisce la selezione attraverso un meccanismo basato su eventi.

```
@Component({
  selector: 'app-did-selector',
  templateUrl: './did-selector.component.html',
  styleUrls: ['./did-selector.component.css']
})
export class DidSelectorComponent implements OnInit, OnDestroy, AfterViewInit {

  private _dids$: Observable<string[]>
  private _err: any

  @Output() select
  selectedDid: string | undefined

  constructor(
    private route: ActivatedRoute,
    private router: Router,
    private didService: DidService,
    ...
  ) {
    // inizializzazione delle variabili
  }

  ...
  selectDid(): void {
    this.select.emit(this.selectedDid)
  }

  deselectDid(): void {
    this.selectedDid = undefined
  }
}
```

Listato 5.18: Frammento di codice del `DidSelectorComponent`.

Il template (Listato 5.19) include un `MatRadioGroup` [6], un elemento che realizza una lista di opzioni selezionabili. Nella logica del `DidSelectorComponent`, mostrata nel Listato 5.18, è contenuto il codice per gestire lo stato del Component che comprende gli elementi visualizzati nell'elenco di opzioni, l'opzione correntemente selezionata, e l'eventuale presenza di errori.

```
1 <mat-radio-group [(ngModel)]="selectedDid" class="radio-group"
  ↪  (change)="selectDid()">
2   <mat-radio-button *ngFor="let did of dids$ ..."
    ↪  [value]="did">{{did}}</mat-radio-button>
3 </mat-radio-group>
```

Listato 5.19: Frammento di template del `DidSelectorComponent`. La direttiva `*ngFor` crea tanti elementi DOM quanti sono gli oggetti dell'array che viene iterato o dello stream di dati sottoscritto (`dids$`).

Lo scambio dei dati da e verso il template, avviene attraverso un meccanismo di binding, offerto da Angular in quattro diverse modalità. Consideriamo il codice presente nel Listato 5.19:

- `(change)="selectDid()"`: event binding. Esegue l'istruzione specificata (`selectDid()`) quando si verifica un evento (`change`).
- `[value]="did"`: property binding. Assegna un valore (`did` contenuto nella variabile definita in `DidSelectorComponent`) ad una proprietà di un elemento HTML (`value`).
- `{{did}}`: string interpolation. Visualizza il risultato di un'istruzione o il valore di una variabile (`did` nel `DidSelectorComponent`) all'interno del template, come una stringa dinamica.
- `[(ngModel)]="selectedDid"`: two-way binding. Crea un flusso bidirezionale di dati tra template e classe `@Component` attraverso una variabile (`selectedDid` definita in `DidSelectorComponent`) e l'utilizzo di una direttiva (`ngModel` definita in `FormsModule` standard di Angular), cioè una classe che aggiunge comportamenti addizionali agli elementi di un template.

Descriviamo quindi il comportamento del `DidSelectorComponent` realizzato attraverso la combinazione dei quattro tipi di binding descritti.

L'elenco dei DID visualizzati dall'utente, comprende dei `<mat-radio-button>` con i valori (`[value]`) che corrispondono ai DID ottenuti dal server, e delle caselle selezionabili accompagnate dal testo specificato da `{{did}}`. Quando viene effettuata una selezione, il `<mat-radio-group>` aggiorna il valore contenuto nel `selectedDid`, grazie alla direttiva `ngModel`, che rende l'elemento un `FormControl` [2], cioè un componente che mantiene sincronizzato il valore contenuto nel `selectedDid` con quello della vista del Component. La selezione allora, scatena l'evento `change` del `<mat-radio-group>`, su cui è in ascolto il metodo `selectDid()`. L'invocazione di questa funzione provoca l'emissione dell'evento `select`, che allo stesso modo di `change`, notifica che è avvenuta una selezione da parte dell'utente. La variabile `select` è un oggetto di tipo `EventEmitter`, che definisce un evento che viene registrato tramite l'annotazione `@Output()`. Le funzioni che rimangono in ascolto dell'evento allora, ricevono anche il valore dell'emissione.

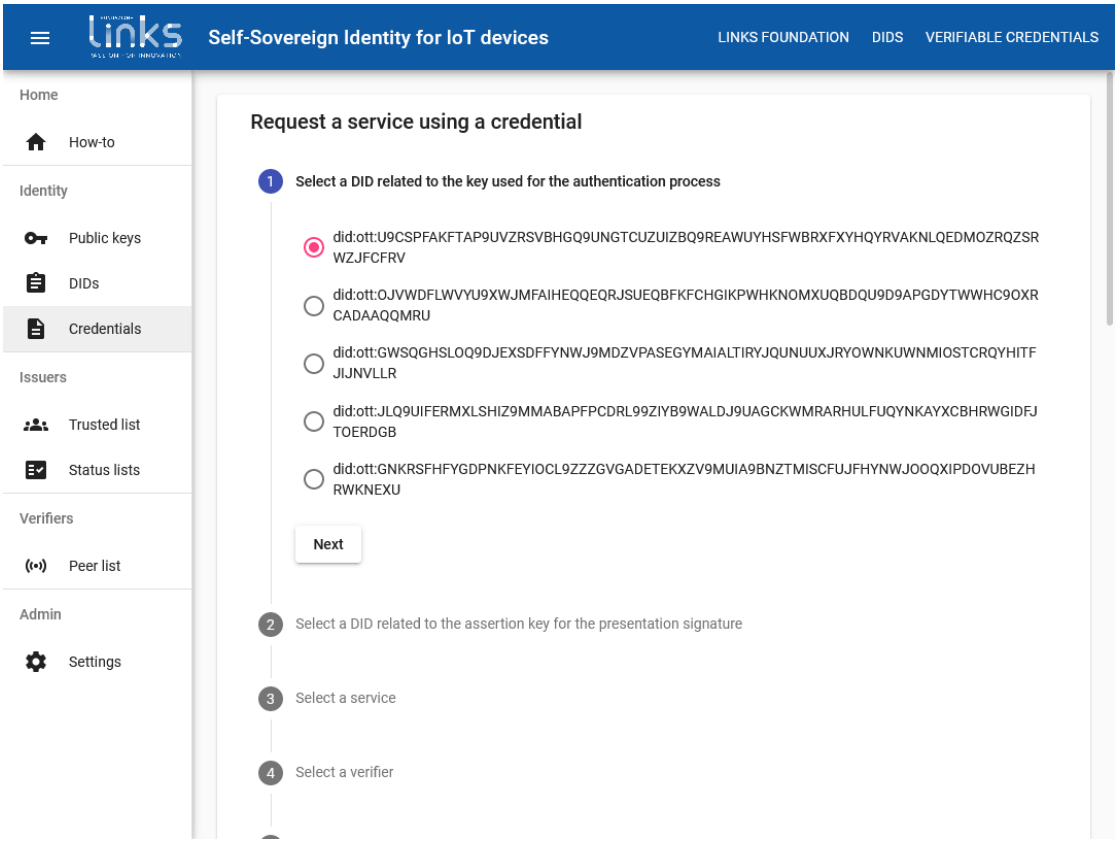


Figura 5.7: Pagina `/device/:owner/credentials/use`.

Grazie alla definizione degli eventi, i Component riescono a creare flussi di dati basati sulle azioni dell'utente. Il comportamento del `DidSelectorComponent` viene riutilizzato in tutte le pagine che richiedono la selezione di un DID, come ad esempio la pagina che si trova all'indirizzo `/device/:owner/credentials/use`, illustrata in Figura 5.7. Il `Crede␣ntialRevokeComponent` incluso in questa pagina contiene un `MatStepper` [8], che definisce una procedura guidata suddivisa in passaggi logici. Tra questi vi è quello della selezione di un DID per il processo di autenticazione, come mostrato nel Listato 5.20. Quando l'utente sceglie una delle opzioni, viene innescato l'evento `select` su cui il `<mat-step>` è in ascolto. L'oggetto `$event`, che corrisponde all'oggetto emesso dall'evento, sarà salvato nella variabile `authDidStep` del `CredentialRevokeComponent`, che completa il passaggio richiesto e sblocca quello successivo nella procedura guidata.

```
1 <mat-step [completed]="authDidStep">
2   ...
3     <app-did-selector (select)="$event ? authDidStep=true : authDidStep=false;
      ↪  selectedAuthDid=$event"
4         #authDs>
```

```
5       </app-did-selector>
6         ...
7   </mat-step>
```

Listato 5.20: Frammento di template del `CredentialRevokeComponent`.

Le pagine per la visualizzazione di Verifiable Credential, DID document e Status List Presentation, offrono la possibilità di visualizzare i contenuti in formato JSON grazie ad uno switch posizionato nelle card che racchiudono il contenuto dei dati (Figura 5.8).

```
1  <div *ngIf="!json_toggle.checked; else json">...</div>
2  ...
3  <mat-slide-toggle #json_toggle>JSON</mat-slide-toggle>
4  ...
5  <ng-template #json>
6    <div [innerHTML]="printJson(credential) | safeStyle" #json></div>
7  </ng-template>
```

Listato 5.21: Frammento di template del `CredentialListComponent` per la visualizzazione di un contenuto in formato JSON.

Il meccanismo di switch permette quindi di cambiare la visualizzazione degli elementi del template in maniera dinamica. Nel `CredentialListComponent` ad esempio, i dati sono contenuti all'interno di elementi `div` a cui vengono assegnati degli identificativi, che indicheranno *quali* elementi rendere attivi sulla base del valore di un `MatSlideToggle` [7], che visualizza un interruttore booleano.

Consideriamo il Listato 5.21, `*ngIf` è una direttiva strutturale di Angular che include un template e un'espressione. Se l'espressione è vera rende visibili gli elementi contenuti nel selettore che specifica la direttiva (`<div>`), altrimenti viene visualizzato un altro elemento, identificato dal nome a cui si riferisce la clausola opzionale `else` (`#json`). Il valore del `MatSlideToggle` è indicato dalla proprietà `checked`, contenente lo stato dello switch (`json_toggle.checked`). Leggendo il valore di questa proprietà all'interno della direttiva `*ngIf`, è possibile attivare e disattivare il contenuto incluso nell'elemento `ng-template`, che risponde ai cambiamenti di stato dello switch. Nella pagina viene quindi visualizzato di default il contenuto dei modelli all'interno di layout responsive, oppure sotto forma di stringa in formato JSON.

L'elemento del template identificato da `#json`, racchiude il suo contenuto in un elemento `div`. `[innerHTML]` implementa un property binding attraverso cui può essere modificato il codice HTML dell'elemento contenuto. Pertanto, sostituendo il codice con il risultato prodotto dalla funzione `printJson`, mostrata nel Listato 5.22, può essere visualizzato un testo costruito con l'elemento `<pre></pre>`, che viene tipicamente utilizzato per lasciare intatti gli spazi bianchi della stringa che contiene, rendendone più fruibile la visualizzazione.

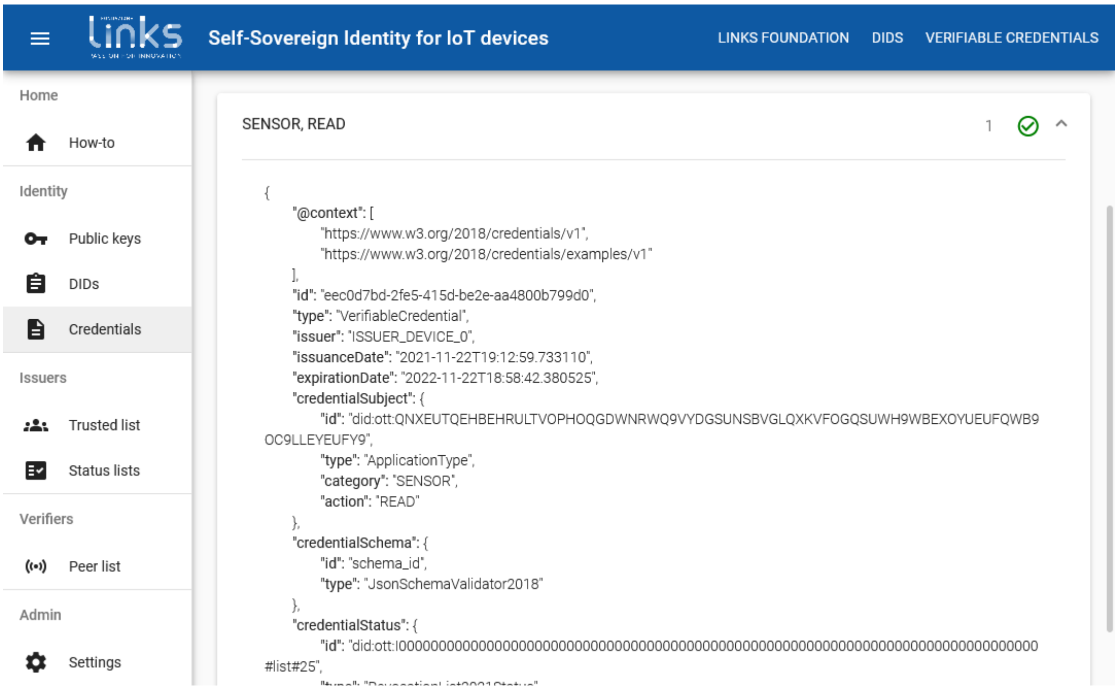
```
printJson(cred: CredentialWithStatus): string {
  ...
  let json = JSON.stringify(c, null, 8)
  let pretty_json = json.replace(/\"\S+\":\s+/g, "<b>$&</b>")
  return `<pre style="margin: 2vw; font-family: inherit; font-weight: 300;
  ↪  white-space: break-spaces; word-break: break-all;">${pretty_json}</pre>`
}
```

Listato 5.22: Funzione di conversione di un oggetto `CredentialWithStatus` in una stringa in formato JSON (classe `CredentialListComponent`).

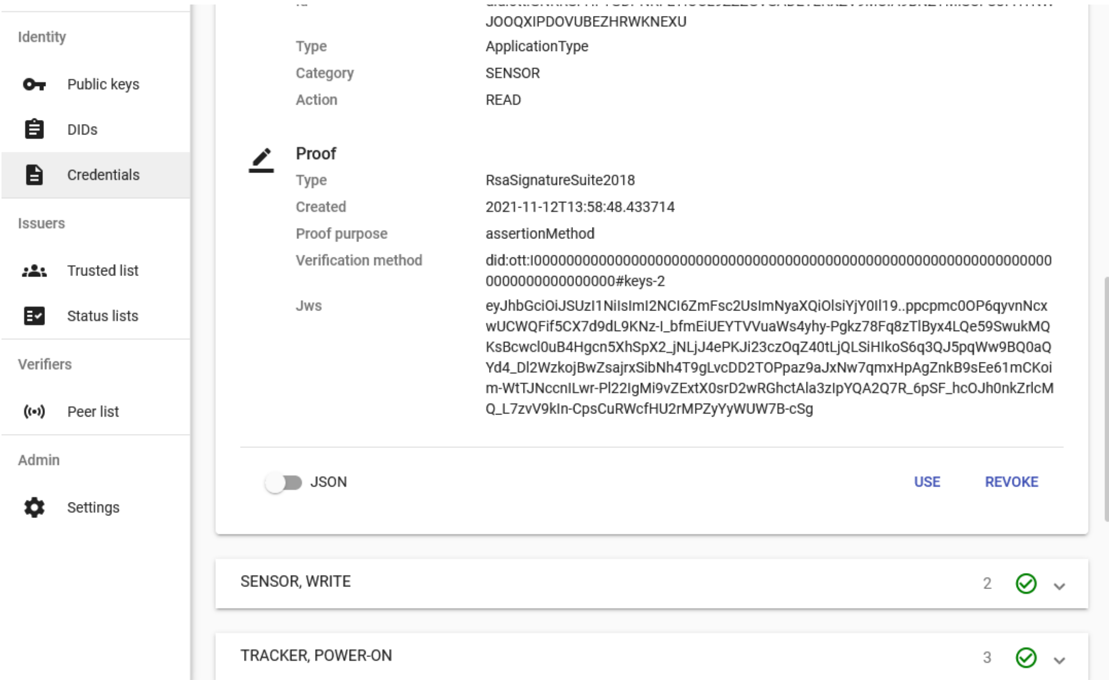
Angular applica delle restrizioni sulle modifiche ai contenuti della pagina, per ragioni di sicurezza. Sostituire il codice HTML in maniera diretta come mostrato nel Listato 5.21 infatti, non è consentito a causa di possibili tentativi di exploit. Per far fronte a questo problema, e lasciare che la pagina visualizzi correttamente il testo prodotto dalla funzione `printJson`, si fa uso delle Pipe. Questo tipo di oggetti messi a disposizione da Angular, tornano utili per trasformare i dati del template su cui vengono applicati. Lo scopo della `SafeStylePipe` (Listato 5.23) è quindi quello di preservare lo stile definito per la stringa racchiusa nell'elemento `<pre></pre>`, prodotta dalla funzione `printJson`. Attraverso l'oggetto `sanitizer` del tipo `DomSanitizer`, viene effettuata la *sanitization* del valore in output al fine di prevenire exploit che fanno uso delle tecniche di Cross Site Scripting. Siccome la stringa visualizzata è sicura, possiamo applicare un bypass sul controllo dello stile, mantenendo intatte le regole del testo per visualizzarne correttamente il contenuto.

```
@Pipe({
  name: 'safeStyle'
})
export class SafeStylePipe implements PipeTransform {
  constructor(private sanitizer: DomSanitizer) { }
  transform(value: any, ...args: unknown[]): unknown {
    return this.sanitizer.bypassSecurityTrustHtml(value);
  }
}
```

Listato 5.23: Definizione di `SafeStylePipe`.



(a) Contenuto di una Verifiable Credential in formato JSON.



(b) Contenuto di una Verifiable Credential.

Figura 5.8: Pagina `/device/:owner/credentials`.

Le Pipe, oltre a definire meccanismi di controllo, si rivelano utili anche per nascondere gli elementi di un template. Il `CredentialListComponent` contenuto nella pagina `/device/:owner/credentials` (Figura 5.9) consente all'utente di filtrare le credenziali visualizzate, mediante l'utilizzo di `MatChip` [4], dei bottoni selezionabili contenenti una stringa e un valore. Questo componente permette di creare gruppi di chip selezionabili in maniera multipla, ognuno dei quali è in grado di emettere un evento non appena ne viene selezionato il valore. Dunque, per filtrare le Verifiable Credential e visualizzare solo quelle che soddisfano le selezioni fatte dall'utente, è necessario definire un filtro e delle espressioni su cui fare un match. La Pipe che implementa il filtro quindi, scarta tutti i valori che non soddisfano le condizioni composte dalle opzioni in cima alla pagina.

```
<ng-container
    *ngIf="(credentials$ | async)! | credentialFilter:
    ↪ revokedFilter:actionFilters:categoryFilters as credentials">
    <div class="chips">
        <mat-chip-list multiple>
            <mat-chip *ngFor="let action of actions" [value]="action"
            ↪ (click)="toggleActionChip(ac)"
                #ac="matChip">
                {{action}}
            </mat-chip>
        </mat-chip-list>
        ...
        <mat-chip-list>
            <mat-chip [value]="false" (click)="toggleStatusChip(vc)"
            ↪ #vc="matChip">
                VALID
            </mat-chip>
            <mat-chip [value]="true" (click)="toggleStatusChip(rc)"
            ↪ #rc="matChip">
                REVOKED
            </mat-chip>
        </mat-chip-list>
    </div>
    ...
</ng-container>
```

Listato 5.24: Frammento di template del `CredentialListComponent` che include i chip per il filtraggio delle Verifiable Credential

Prendiamo in considerazione il Listato 5.24. La proprietà `value` contiene il valore da passare alla funzione `toggleActionChip` del `CredentialListComponent`. Quando lo stato del chip selezionato viene aggiornato, cioè ad ogni selezione, il valore emesso (`value`) sarà inserito in un array utilizzato per il match delle condizioni del filtro. La stessa logica è applicata al secondo gruppo di chip, contenente le categorie di dispositivi, e in maniera esclusiva ai due chip che filtrano le credenziali sulla base del loro stato, valide o revocate.

```
private _actionFilters
private _categoryFilters
private _revokedFilter: boolean | undefined

...
toggleStatusChip(c: MatChip) {
  c.toggleSelected()
  this._revokedFilter = this._revokedFilter === c.value ? undefined : c.value
}

toggleCategoryChip(c: MatChip) {
  this._categoryFilters = this.selectChip(c, this._categoryFilters)
}

toggleActionChip(c: MatChip) {
  this._actionFilters = this.selectChip(c, this._actionFilters)
}

selectChip(c: MatChip, filterArr: any[]): any[] {
  c.toggleSelected()
  if (c.selected) {
    filterArr.push(c.value)
  } else {
    filterArr.splice(filterArr.indexOf(c.value), 1)
  }
  return filterArr.slice()
}
```

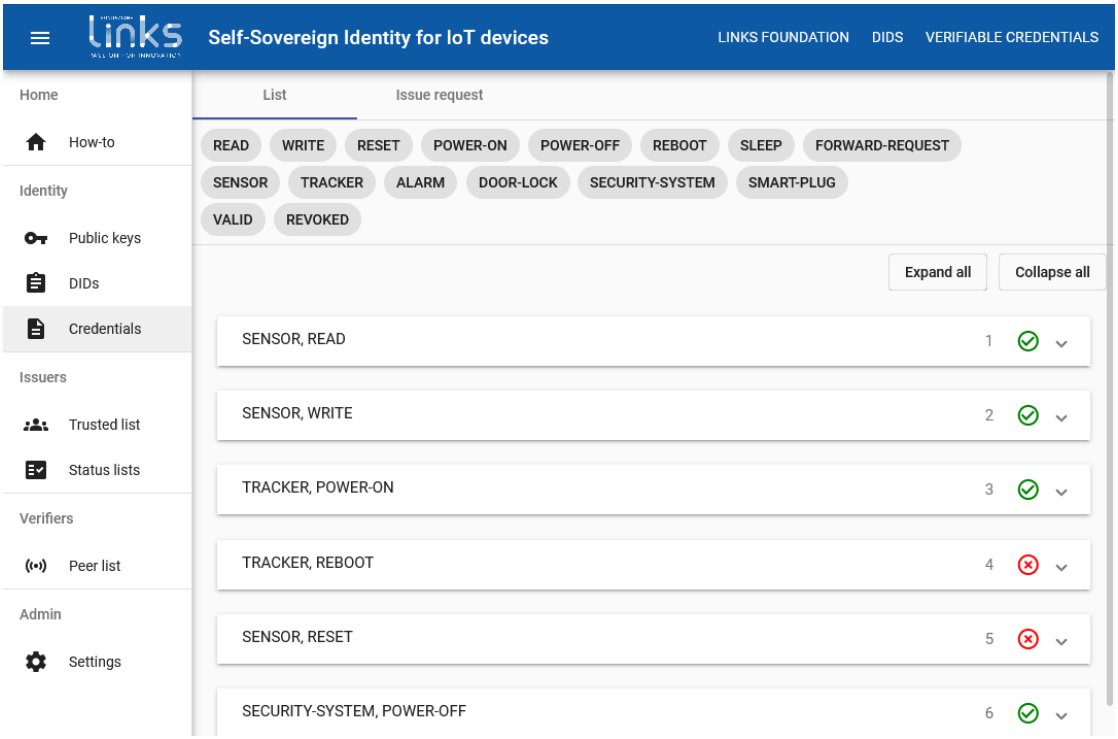
Listato 5.25: Funzioni per la gestione dei `MatChip` all'interno del template (classe `CredentialListComponent`).

Tutti i valori selezionati nei chip vengono quindi emessi e salvati all'interno di tre array, quelli denominati con il suffisso `Filter`, come vediamo nel Listato 5.25. Il filtro è implementato nella classe `CredentialFilterPipe`, che accetta un array di `CredentialWithStatus`, cioè un'interfaccia che contiene una Verifiable Credential e relativo status corrente, e i tre array considerati prima. La funzione `transform` quindi, utilizza il metodo standard `filters` e scarta tutte le credenziali che non soddisfano le tre condizioni specificate. Il risultato della Pipe provoca un refresh della vista del `CredentialListComponent`, che aggiornerà l'elenco delle credenziali visualizzando solo quelle filtrate. Ogni volta che l'utente cambia la selezione dalle opzioni in cima alla pagina, viene lanciata la `transform` della `CredentialFilterPipe`, così da applicare nuovamente il filtro ai contenuti.

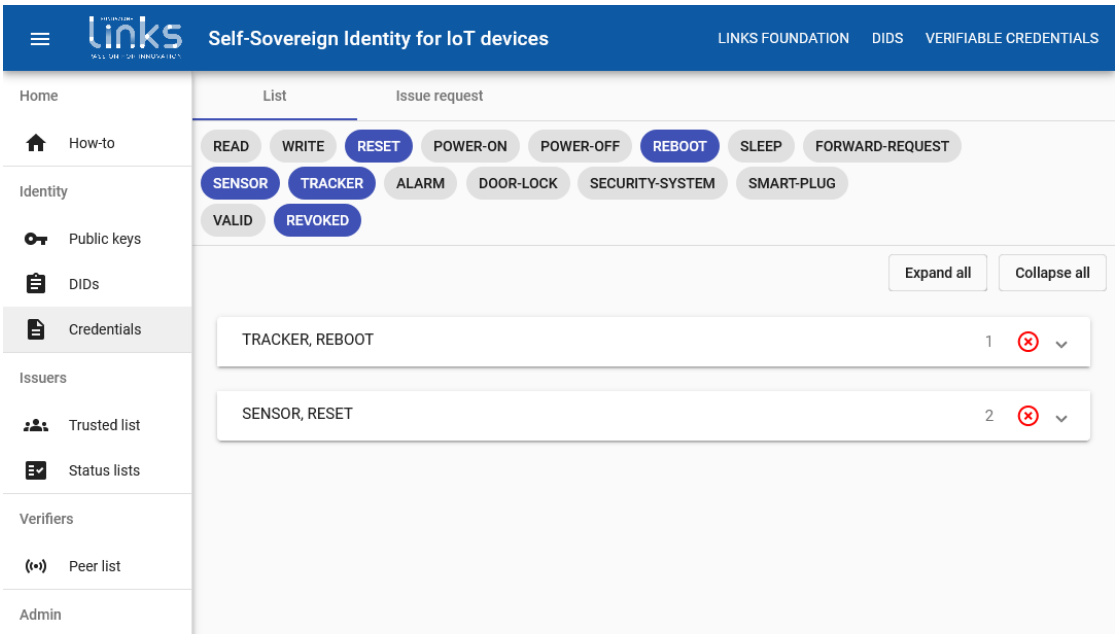
```
@Pipe({name: 'credentialFilter'})
export class CredentialFilterPipe implements PipeTransform {
```

```
3     transform(
4       credentials: CredentialWithStatus[],
5       revokedFilter: boolean | undefined,
6       actionFilters: string[],
7       categoryFilters: string[]
8     ): CredentialWithStatus[] {
9       if (!credentials) {
10        return []
11      }
12      return credentials.filter(
13        credential => {
14          const rFilter = revokedFilter !== undefined ? (revokedFilter ?
            ↪  credential.revoked === true : credential.revoked === false) : true
15          const aFilter = actionFilters.length ?
            ↪  actionFilters.includes(credential.credentialSubject.action) : true
16          const cFilter = categoryFilters.length ?
            ↪  categoryFilters.includes(credential.credentialSubject.category) :
            ↪  true
17          return rFilter && aFilter && cFilter
18        }
19      )
20    }
21  }
```

Listato 5.26: Implementazione della Pipe per filtrare Verifiable Credential.



(a) Elenco Verifiable Credentials.



(b) Filtro elenco.

Figura 5.9: Pagina `/device/:owner/credentials`.

5.4.4 Richieste di rete e flusso di dati

Nelle applicazioni sviluppate in Angular, i Service rappresentano degli oggetti che forniscono funzionalità comuni. Tipicamente, vengono utilizzati per implementare la logica per gestire lo stato dei modelli e per richiedere i dati al back-end. Angular crea le istanze dei Service tramite gli Injector, che vengono invocati specificando nel costruttore del Component il tipo di Service richiesto. Per definire il Provider con cui l'Injector fornisce le istanze, è sufficiente annotare la classe del Service con `@Injectable()`, e specificare *chi* avrà questa responsabilità. Il `CredentialService` ad esempio, viene iniettato dal modulo root (Listato 5.27), e di conseguenza risulta disponibile per tutti i Component dell'applicazione.

La relazione che intercorre tra Service e Provider fa sì che il ciclo di vita dei Service, sia legato al modulo che si occupa del suo provisioning. Ciò significa che un Service può *sopravvivere* alla distruzione del Component in cui è utilizzato. Di conseguenza, eventuali task lanciati in background dal Service potrebbero causare un consumo di risorse inutili se non venissero arrestati.

```
1  @Injectable({providedIn: 'root'})
2  export class CredentialService {}
3
4  @Component({
5    selector: 'app-credential-list',
6    templateUrl: './credential-list.component.html',
7    styleUrls: ['./credential-list.component.css']
8  })
9  export class CredentialListComponent implements OnInit, AfterViewInit {
10   constructor(
11     private credentialService: CredentialService,
12     ...
13   ) {...}
```

Listato 5.27: Costruttore di `CredentialListComponent` che richiede un'istanza del `CredentialService` tramite dependency injection.

I dati da visualizzare nelle pagine sono descritti da interfacce che rappresentano i modelli gestiti dal back-end. I Service che si occupano delle chiamate di rete, verso il server che espone le API REST, collezionano i dati per popolare gli elementi contenuti nei template, attraverso dei flussi gestiti in maniera asincrona. Le risposte ottenute devono essere processate, talvolta arricchite e aggiornate, in base alle azioni compiute dall'utente. La scelta di adottare la libreria RxJS [51] è dovuta quindi all'efficienza con cui è possibile definire delle pipeline per la trasformazione dei dati, secondo una logica asincrona ed event-based. Grazie alle nuove strutture introdotte e alla possibilità di combinare le pipeline, RxJS è diventata una delle librerie più utilizzate in ambito di sviluppo web.

I concetti introdotti sono i seguenti:

- Observable: rappresenta l'idea di una collezione di dati futuri o di eventi.

- Observer: callback in ascolto dei valori emessi dagli Observable.
- Subscription: rappresenta l'esecuzione di un Observable.
- Operator: funzioni pure che permettono di gestire flussi di dati secondo uno stile di programmazione funzionale.
- Subject: Observable speciali che emettono valori in multicast.
- Scheduler: dispatcher che coordina come avvengono le operazioni.

In altre parole, un Observable è un oggetto che consente di creare flussi di dati manipolati dagli Operator, attraverso la definizione di pipeline su cui viaggiano i valori emessi. L'emissione avviene alla sottoscrizione dell'Observable, che come un iteratore, produce dati in sequenza. Le callback registrate al momento della sottoscrizione, cioè gli Observer, ricevono i dati emessi dall'Observable fino. I Subject invece, emettono flussi di dati come gli Observable, ma consentono anche la sottoscrizione, e tipicamente vengono utilizzati per emettere flussi di dati in multicast, cioè ricevuti da molteplici Observer. Infine, durante la sottoscrizione di un Observable, è possibile registrare una funzione per i tre tipi di evento: emissione dei dati in corso, di errore e di completamento del flusso.

In sostanza, RxJS introduce uno stile di programmazione detto reattivo, basato sui pattern observer e iterator, grazie alla definizione degli Observable, che sono perfetti per gestire i flussi di dati provenienti dal server.

Il servizio `HttpClient` offerto da Angular, si occupa di lanciare le richieste di rete sotto forma di `Observable`, che di conseguenza andranno ad emettere i dati ricevuti da visualizzare nella pagina. Essendo il Typescript un linguaggio tipizzato, i metodi che ricevono i dati dal back-end, e che fluiscono verso i Component, si presentano in una forma già compatibile con quella attesa dal template.

Analizziamo il Listato 5.28 che mostra un esempio di richiesta al server. L'Observable ritornato dalla `get` dell'`HttpClient` specifica il tipo di oggetto (modello) che sarà emesso. La pipeline definita sul valore di ritorno inoltre, fa uso dell'Operator `catchError` che da la possibilità di gestire le chiamate di rete andate in errore, sostituendo l'Observable atteso con uno che emetterà un valore utile a notificare l'utente del problema. `throwError` pertanto, scatenerà l'emissione di un messaggio di errore, che di conseguenza sarà visualizzato al posto del contenuto atteso, grazie alla dinamicità del template. Il Component non deve quindi fare altro che richiamare il Service, invocare la funzione che fa la richiesta al server e sottoscrivere all'Observable ritornato, registrando le callback per ogni tipo di evento.

```
1  // Service
2  getStatusLists(): Observable<StatusListPresentation[]> {
3    return this.http.get<StatusListPresentation[]>(
4      `${this.API}/status-lists`
5    ).pipe(
6      catchError(() => throwError("Unable to fetch status lists"))
7    )
8  }
```

```
9
10 // Component
11 this.issuerService.getStatusLists().subscribe(
12   statusLists => {...},
13   error => { this._err = err; return EMPTY },
14   () => { console.log("completed") }
15 )
```

Listato 5.28: Funzione per ottenere le status list dal server (classe `IssuerService`) e sottoscrizione all'Observable frutto della richiesta (classe `StatusListPresentationCompon⌋ent`).

Sottoscrivere ad un Observable vuol dire quindi avviare uno stream di dati, indipendente dal ciclo di vita del Component in cui è avvenuta la sottoscrizione. Come accennato precedentemente, questo comportamento costituisce fonte di errore e facilita l'insorgere di problemi di memory leak, se la sottoscrizione non viene gestita correttamente. La tecnica più utilizzata, è quella di fare uso della Pipe `async`, che si occupa di creare una sottoscrizione direttamente nel template del Component. Alla creazione della view quindi, la Pipe si sottoscrive all'Observable, e in automatico si disiscrive non appena il Component viene distrutto. È pratica comune allora quella di invocare il Service all'inizializzazione del Component, creare un Observable che emetterà i dati ottenuti dal server, e salvarlo in una variabile privata (`_actions$`[3]), come mostrato nel Listato 5.29. Solitamente questi Observable vengono esposti dal Component in sola lettura, tramite il costrutto `get`, al fine di proteggere la sorgente dei dati emessi.

```
1  private _err: any
2  private _actions$: Observable<string[]>
3
4  constructor(
5    private credentialService: CredentialService
6  ) {
7    this._actions$ = this.credentialService.getActionTypes().pipe(
8      tap(() => { this._err = undefined }),
9      catchError(err => { this._err = err; return EMPTY })
10   )
11 }
12
13 get actions$() {
14   return this._actions$
15 }
16
```

[3]Per convenzione le variabili di tipo Observable sono chiamate ponendo un simbolo $ alla fine del nome.

```
17  get err() {
18    return this._err
19  }
```

Listato 5.29: Istruzioni del `ActionSelectorComponent` per effettuare una richiesta di rete tramite il `CredentialService` ed ottenere i dati da visualizzare nel template.

La Pipe `async` torna utile per realizzare anche altri meccanismi, che sfruttano la dinamicità dei template in cui viene utilizzata. Il codice presente nel Listato 5.30 mostra il template dell'`ActionSelectorComponent`, che consente all'utente di selezionare il tipo di azione da includere nella Verifiable Credential rilasciata dall'Issuer. La sottoscrizione all'Observable `actions$` avviene attraverso la direttiva `*ngIf`, che legge come valore iniziale `undefined`. L'effetto di combinare la Pipe con questa direttiva, è quello di visualizzare l'elemento specificato nella clausola `else` fino a quando lo stream dell'Observable non entra nello stato di emissione. Con questa tecnica si può allora dare all'utente l'impressione del *caricamento*, mediante la visualizzazione temporanea di uno spinner (`MatSpinner` [5]). Nel caso in cui la richiesta di rete fatta all'inizializzazione del Component non va a buon fine, il messaggio di errore emesso dall'Observable sarà inserito nella variabile `err` e sarà visualizzato al termine del caricamento, cioè quando la richiesta fatta dal Service viene completata.

```
1  <div *ngIf="actions$ | async as actions; else loadingOrError">
2      <mat-radio-group [(ngModel)]="selected" class="radio-group"
       ↪  (change)="choose()">
3          <mat-radio-button *ngFor="let action of actions"
           ↪  [value]="action">{{action}}</mat-radio-button>
4      </mat-radio-group>
5  </div>
6  <ng-template #loadingOrError>
7      <span *ngIf="err; else loading">{{err}}</span>
8      <ng-template #loading>
9          <mat-spinner></mat-spinner>
10     </ng-template>
11 </ng-template>
```

Listato 5.30: Frammento di template del `ActionSelectorComponent`.

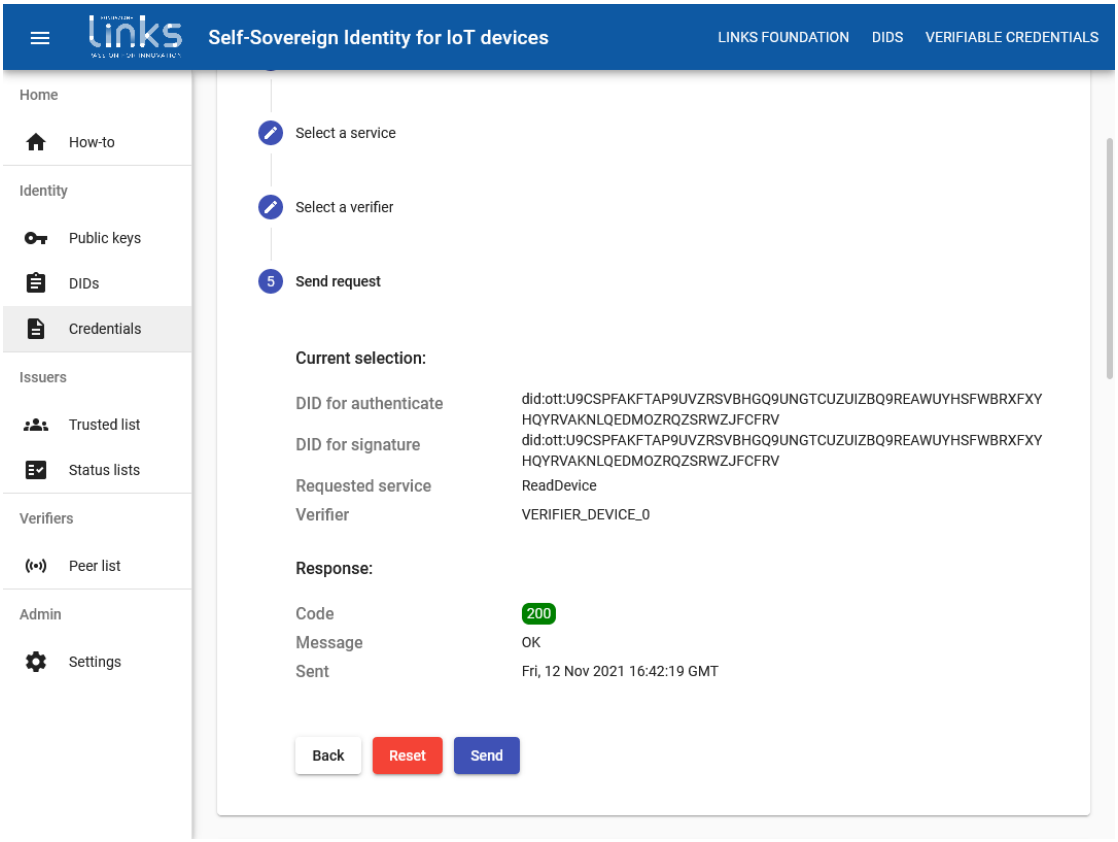
Figura 5.10 mostra un'altra pagina, che si basa sull'utilizzo delle pipeline per visualizzare correttamente i messaggi di errore ritornati dal back-end. Nel Listato 5.31 è contenuta l'implementazione del metodo `useCredential`, appartenente al `CredentialService`, per inviare una richiesta POST al server e accedere al servizio di un Verifier. La pipeline registrata sull'Observable ritornato dall'`HttpClient`, contiene la funzione `handleErrorRes ponse`, che si occupa di catturare gli errori di tipo `HttpErrorResponse`, lanciati quando la

richiesta di rete fallisce. Il risultato della chiamata però, sarà in ogni caso un Observable che emetterà lo stesso tipo di dato atteso dal template, ma che stavolta contiene un messaggio che segnala un errore. Per questo motivo, anche a fronte di accessi non autorizzati, la risposta sarà comunque visualizzata correttamente nel layout del template.

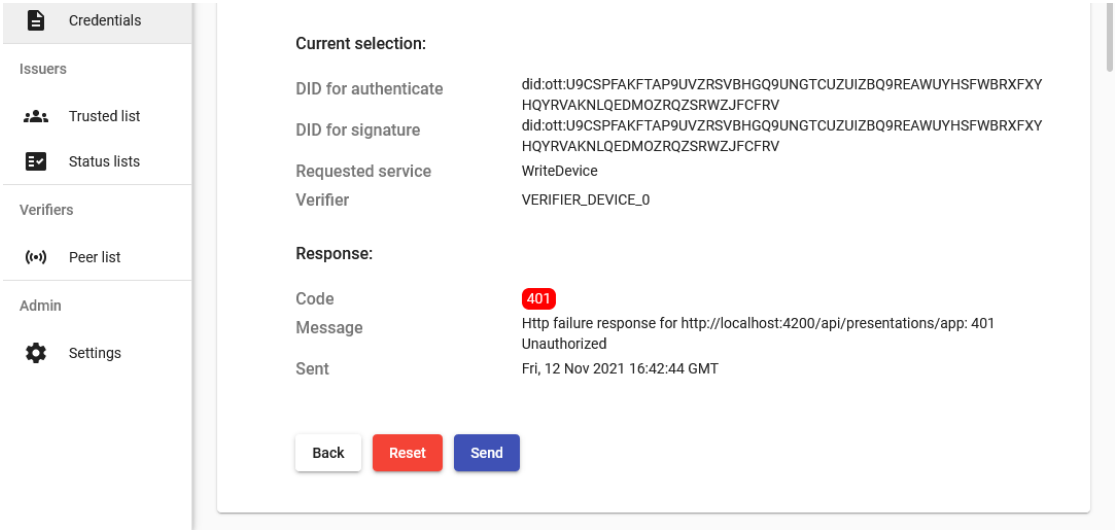
```
useCredential(
  credentialId: string,
  authCryptoId: string,
  assertCryptoId: string,
  verifierId: string,
  serviceType: string,
  owner: string
): Observable<HttpResponse<string>> {
  return this.http.post<string>(
    `${this.API}/presentations/app`,
    {
      credentialId,
      authCryptoId,
      assertCryptoId,
      verifierId,
      serviceType,
      owner
    },
    { observe: "response" }
  ).pipe(
    catchError(err => this.handleErrorResponse(err))
  )
}

handleErrorResponse(err: HttpErrorResponse): Observable<HttpResponse<any>> {
  return of(new HttpResponse<any>({
    status: err.status,
    statusText: err.message
  }))
}
```

Listato 5.31: Funzione del `CredentialService` per l'invio delle richieste di accesso ad un servizio.



(a) Accesso al servizio di un Verifier.



(b) Accesso non autorizzato.

Figura 5.10: Pagina `/device/:owner/credentials/:id/use`.

5.4.5 Aggiornamento dei contenuti

Quando avviene la navigazione verso una nuova pagina, le richieste di rete vengono lanciate non appena la creazione dei Component viene completata. Durante l'inizializzazione, viene invocato il servizio Angular `ActivatedRoute` per ottenere la mappa dei parametri contenuti nell'URL verso cui è avvenuta la navigazione. Una volta che tutte le informazioni necessarie vengono lette dal path, il Service è in grado di lanciare le richieste al server, per ricevere le informazioni aggiornate sui contenuti da mostrare. Le risposte alle richieste vengono quindi gestite attraverso le pipeline registrate sugli Observable, a cui il Component sottoscrive.

Dato che il comportamento appena descritto è attivato tutte le volte che l'utente completa la navigazione verso una pagina dedicata alla presentazione di contenuti, bisogna porre molta attenzione all'utilizzo degli Operator che formano le pipeline degli Observable. Le sottoscrizioni infatti, non essendo legate al ciclo di vita dei Component, continuano la loro esecuzione anche all'uscita della pagina da cui vengono create. Di conseguenza, bisogna evitare che sottoscrizioni indesiderate consumino risorse, e liberare quelle per le richieste completate. In caso di navigazioni ripetute, l'utente potrebbe infatti causare l'avvio di task inutili che andrebbero interrotti.

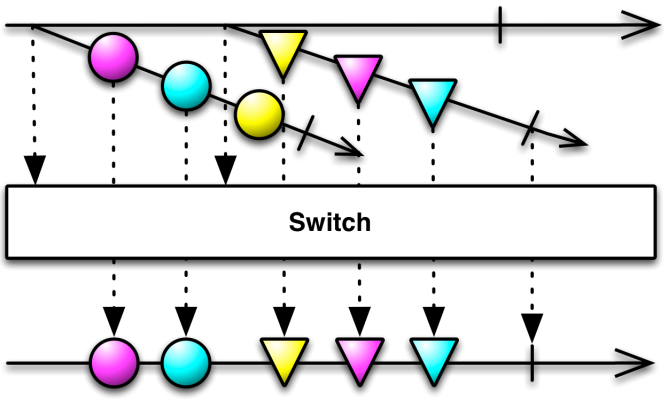


Figura 5.11: Diagramma marble dell'operatore `switchMap`.

Introduciamo quindi un Operator di utilizzo comune: `switchMap`. Per definizione, l'operatore switchMap accetta in input un Observable e produce in output un unico stream di dati. Lo stream in uscita è il risultato di un'operazione di flattening, che produce un flusso che emette i valori delle sorgenti interne (inner Observable) a cui eventualmente viene applicata una funzione di trasformazione. La particolarità dello switch è data dal modo in cui le sottoscrizioni delle sorgenti in input vengono gestite. Infatti, quando cambia la fonte dei dati emessi in input, la sottoscrizione a quella precedente viene completata, causando il rilascio delle risorse dedicate a favore di quella nuova. Il comportamento desiderato per le richieste di rete è quello di elaborare solo quelle correnti, evitando di completare quelle precedenti, ormai prive di rilevanza e ridondanti. L'operatore switchMap quindi, si presta perfettamente per questo tipo di esigenza.

Listato 5.32 mostra un esempio di come, all'inizializzazione del `CredentialListCompo` ⌋ `nent`, i dati della route della pagina vengono estratti per richiedere al `CredentialService`

di ottenere le informazioni dal server. Ottenuto l'identificativo dell'`owner` (inteso come dispositivo), si ottiene un Observable tramite `getCredentialList`, che emette la lista delle credenziali da lui possedute, frutto di una richiesta GET al server. Notiamo quindi l'utilizzo della `switchMap`, che interrompe le vecchie richieste non appena avviene una seconda navigazione sulla pagina, sostituendo il flusso precedente con quello corrente.

```
ngOnInit(): void {
  this.route.data.pipe(
    switchMap(data =>
    ↪  this.credentialService.getCredentialList(data.owner).pipe(
      mapTo(data.owner)
    )),
    switchMap(owner => this.credentialService.credentialList$.pipe(
      switchMap(credList =>
      ↪  this.credentialService.getCredentialWithStatusList(credList, owner))
    ))
  ).subscribe(
    () => { this._err = undefined },
    err => { this._err = err }
  )

  this.credentialService.getActionTypes().pipe(
    tap(actions => { this._actions = actions })
  ).subscribe()

  this.credentialService.getCategoryTypes().pipe(
    tap(categories => { this._categories = categories })
  ).subscribe()
}
```

Listato 5.32: Codice di inizializzazione del `CredentialListComponent`.

Aggiornare i contenuti di una pagina significa lanciare nuove richieste verso il server per ottenere le nuove risorse. Nelle single-page application però, l'aggiornamento può essere gestito in maniera più conveniente.

All'interno dei Service vengono gestiti degli oggetti che rappresentano lo stato dei modelli salvati sul server. Per ogni risposta ricevuta, il Service provvede ad aggiornarli, e quindi a scatenare un refresh dei dati visualizzati nelle pagine. Tale approccio ha un duplice vantaggio:

1. Il numero di richieste viene minimizzato: dopo aver completato la richiesta, non è necessario effettuare una seconda richiesta per richiedere le risorse aggiornate al server.

2. I contenuti delle pagine vengono automaticamente aggiornati: grazie allo stile di programmazione reattiva, attraverso le pipeline, gli Operator definiscono operazioni collaterali per attivare l'aggiornamento dei contenuti.

I flussi di dati verso i Component, scatenano l'invocazione di istruzioni che hanno il compito di raccogliere le informazioni ottenute e di notificare che il modello è stato aggiornato, sia in caso di successo che di errore. Essendo i Component in ascolto su questi flussi, la view viene automaticamente aggiornata con i nuovi dati ottenuti dal server, senza l'intervento da parte dell'utente. Questo pattern appena descritto realizza quindi un tipo di Service definito cached service, in cui i Component in ascolto rappresentano gli observer.

Prendiamo in considerazione il Listato 5.33, che mostra il codice per costruire un cached service. Per comprenderne il funzionamento è necessario introdurre un tipo di Subject chiamato BehaviourSubject. La differenza tra un BehaviourSubject ed un Subject è data dalla proprietà del primo di emettere l'ultimo valore del flusso non appena avviene la sottoscrizione. Al contrario, per ricevere il valore emesso da un semplice Subject, è necessario creare la sottoscrizione *prima* dell'emissione, mentre invece un Observer può ricevere l'ultimo valore emesso da un BehaviourSubject anche se la sottoscrizione avviene *dopo* l'ultima emissione.

Nel Service vengono esposti degli Observable *derivati* dai BehaviourSubject, cioè delle variabili `readonly` a cui i Component sottoscrivono e che rappresentano la fonte dei dati.

```
1  private _credentialList$
2  private _credentialWithStatusList$
3  private _credentialDataStore
4  private _loading$
5  private _loadingSl$
6  readonly credentialList$
7  readonly credentialWithStatusList$: Observable<CredentialWithStatus[]>
8  readonly loading$
9  readonly loadingSl$
10
11 constructor(private http: HttpClient) {
12   this._credentialList$ = new BehaviorSubject<Credential[]>([])
13   this._credentialWithStatusList$ = new
     ↪  BehaviorSubject<CredentialWithStatus[]>([])
14
15   this._credentialDataStore = [] as Credential[]
16
17   this.credentialList$ = this._credentialList$.asObservable()
18   this.credentialWithStatusList$ =
     ↪  this._credentialWithStatusList$.asObservable()
19
20   this._loading$ = new BehaviorSubject<boolean>(false)
21   this._loadingSl$ = new BehaviorSubject<boolean>(false)
22
23   this.loading$ = this._loading$.asObservable()
24   this.loadingSl$ = this._loadingSl$.asObservable()
25 }
```

Listato 5.33: Codice di inizializzazione del `CredentialService`. Le variabili `_loading*$` si occupano di notificare il Component di richieste in corso.

Il flusso viene creato all'inizializzazione del Component tramite i metodi esposti dal Service, che effettuano il fetching dei modelli dal server e su cui sono costruite le pipeline (`getCredentialList` nel Listato 5.32). Quando i nuovi dati fluiscono attraverso le pipeline, gli Operator si occupano di aggiornare gli oggetti `DataStore` del Service, in cui viene salvato lo stato dei modelli. I `DataStore` rappresentano quindi degli oggetti cached, che ad ogni richiesta al server vengono aggiornati in maniera opportuna. Ma grazie ai BehaviourSubject, viene provocata l'emissione del modello aggiornato mediante una `next`, tipica degli iteratori, che causa il refresh delle view della pagina.

I BehaviourSubject vengono definiti hot Observable, nel senso che rappresentano una sorgente di dati che emette valori a prescindere dalla sottoscrizione. Al contrario, i cold Observable costituiscono una fonte di dati che inizia l'emissione solo dopo la sottoscrizione. Per cui, i Component che si sottoscrivono al cached service, ottengono immediatamente l'ultimo modello emesso, o quello con cui è stato inizializzato. Per evitare effetti indesiderati e gestire correttamente i flussi dati hot allora, è necessario impiegare operatori come `swi⌋ tchMap`, per evitare di lanciare richieste di rete a ripetizione e quindi saturare il client che accede all'applicazione.

Nel Listato 5.32 è mostrato un esempio significativo, su come aggiornare la view di un Component automaticamente, gestendo correttamente la sottoscrizione alla fonte di dati (il cached service).

Attraverso la combinazione di due `switchMap`, la prima fonte rappresentata dal cached service può essere utilizzata per costruire altre risorse da visualizzare nella pagina. Con questo tipo di sottoscrizione, il Component crea quindi la seguente catena di richieste:

1. Viene lanciata una prima richiesta al server, la cui risposta è emessa da un Observable, a cui il Component sottoscrive.

2. La pipeline costruita sul primo Observable ritornato dal Service aggiorna gli oggetti `DataStore`, con i modelli che vengono emessi tramite i `BehaviourSubject`.

3. Il modello emesso causa una seconda operazione, che arricchisce con nuove informazioni le risorse.

4. Angular aggiorna la view dei Component nella pagina.

In alcune view dell'applicazione quindi, ottenere *solo* i modelli non è sufficiente. Ad esempio, nella pagina `/device/:owner/credentials` viene creato un elenco di Verifiable Credential con un informazione aggiuntiva, estendendo l'interfaccia che ne descrive il modello. Le richieste vengono fatte su due endpoint differenti, ma il risultato finale viene combinato in un modello singolo, derivato da quello base (`Credential`), ma contenente anche lo status corrente della credenziale (`CredentialWithStatus`).

Listato 5.34 mostra l'implementazione dell'operazione che avviene all'emissione del primo modello nella pipeline appena descritta. `getCredentialWithStatusList` viene invocato dal Component non appena viene ottenuto dal server l'array delle `Credential` (`credentials`),

da cui viene creato un nuovo Observable tramite l'operatore `from`. I valori emessi creano flussi multipli su cui avviene un'operazione di flattening applicata dall'operatore `mergeMap` (Figura 5.12).

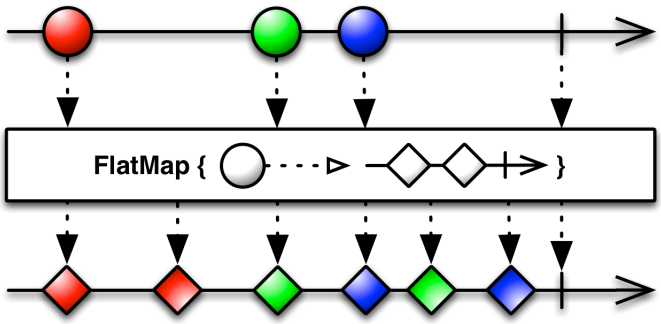


Figura 5.12: Diagramma marble dell'operatore `mergeMap` (o `flatMap`).

Per definizione, `mergeMap` produce in output un flusso singolo su cui vengono emessi i valori degli Observable in input, dopo aver applicato una funzione di trasformazione. La trasformazione in questione, è specificata dall'operatore `map` nella pipeline costruita sul flusso emesso dalla `mergeMap`, finalizzata a creare un oggetto del tipo `CredentialWithStatus`, per ogni `Credential` emessa. Lo status viene quindi richiesto al server dalla `getCredentialStatus` ad ogni emissione. I nuovi oggetti vengono collezionati in un singolo array (`toArray`) che rappresenterà il modello aggiornato, emesso dal `BehaviourSubject` mediante l'operatore `tap`, che permette di eseguire nelle pipeline operazioni trasversali. In presenza di errori, il `catchError` emetterà un messaggio di errore che sarà visualizzato non appena il flusso viene completato.

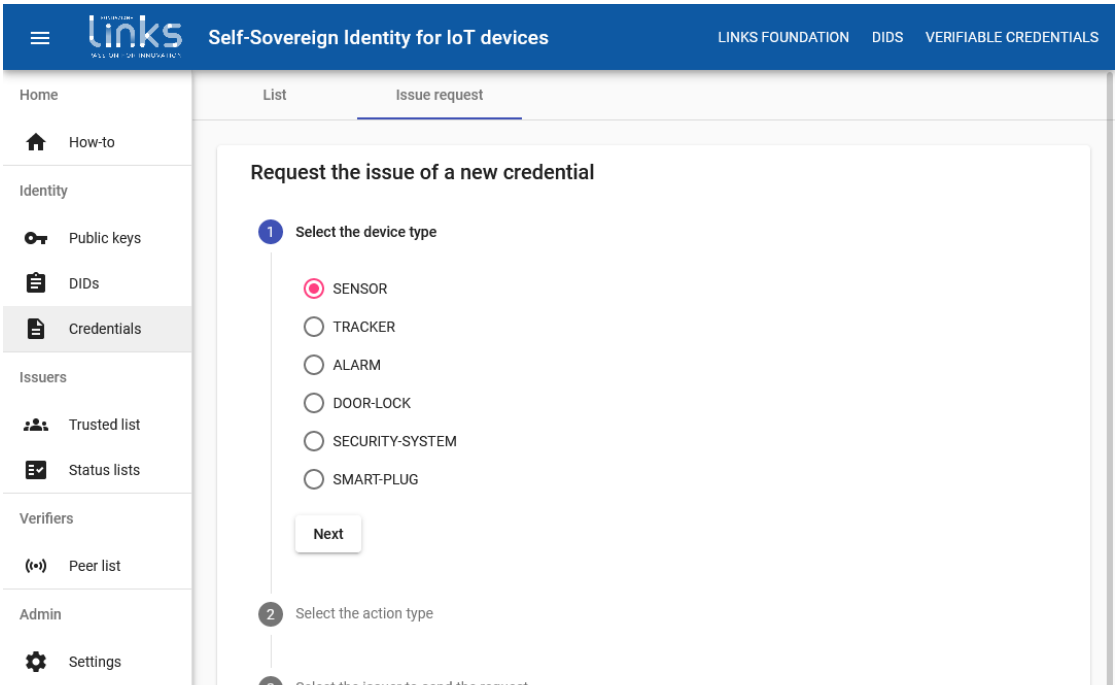
In questo modo, attraverso richieste fatte in parallelo, le pagine vengono corredate con informazioni aggiuntive, riutilizzando gli oggetti emessi dal cached service per creare nuovi modelli. La pagina mostrata in Figura 5.9 ad esempio, consente all'utente di visualizzare la lista delle credenziali (`CredentialWithStatus`) possedute da un dispositivo. Non appena il server risponde ad una nuova richiesta di rilascio, la lista visualizzata nella pagina sarà automaticamente popolata con le nuove credenziali, senza l'intervento da parte dell'utente.

Lo stesso meccanismo è applicato alla lista degli Issuer noti, nella pagina `/device/:owner/credentials` da cui si può richiedere il rilascio di una nuova credenziale (Figura 5.13). Al passaggio di selezione del peer a cui inviare la richiesta, viene visualizzato un elenco aggiornato contenente delle icone che raffigurano lo stato di attivazione degli Issuer. Oppure, lo stesso accade nelle pagine che visualizzano la lista delle chiavi pubbliche (Figura 5.14) e dei DID document del dispositivo (Figura 5.6), che vengono automaticamente aggiornate quando il server completa la creazione dei modelli sul Tangle.

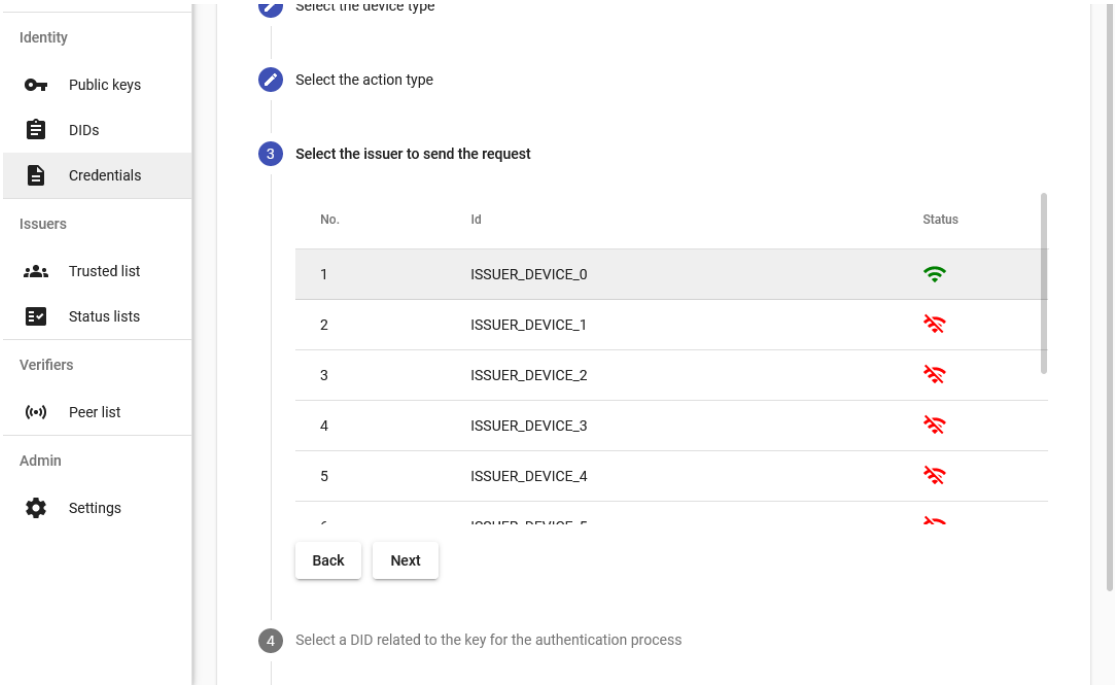
```
1 getCredentialWithStatusList(credentials: Credential[], owner: string):
  ↪ Observable<CredentialWithStatus[]> {
2
3     this._loadingSl$.next(true)
```

```
4       return from(credentials).pipe(
5         mergeMap(cred => this.getCredentialStatus(cred, owner).pipe(
6           map(status => {
7             return {
8               ...cred,
9               revoked: status < 0 ? undefined : status === 1
10            } as CredentialWithStatus
11          })
12        )),
13        toArray(),
14        tap(credList => {
15          this._credentialWithStatusList$.next(credList)
16          this._loadingSl$.next(false)
17        }),
18        finalize(() => { this._loadingSl$.next(false) }),
19        catchError(() => {
20          const err = "Unable to fetch credential with status list"
21          this._credentialWithStatusList$.error(err)
22          return throwError(err)
23        }),
24      )
25    }
```

Listato 5.34: Implementazione della catena per aggiornare le Verifiable Credential nella pagina `/device/:owner/credentials`. La variabile `_loadingSl$` serve per notificare il Component in ascolto del `CredentialService` che la richiesta di rete sta avvenendo.



(a) Selezione dispositivo.



(b) Selezione Issuer.

Figura 5.13: Pagina `/device/:owner/credentials`.

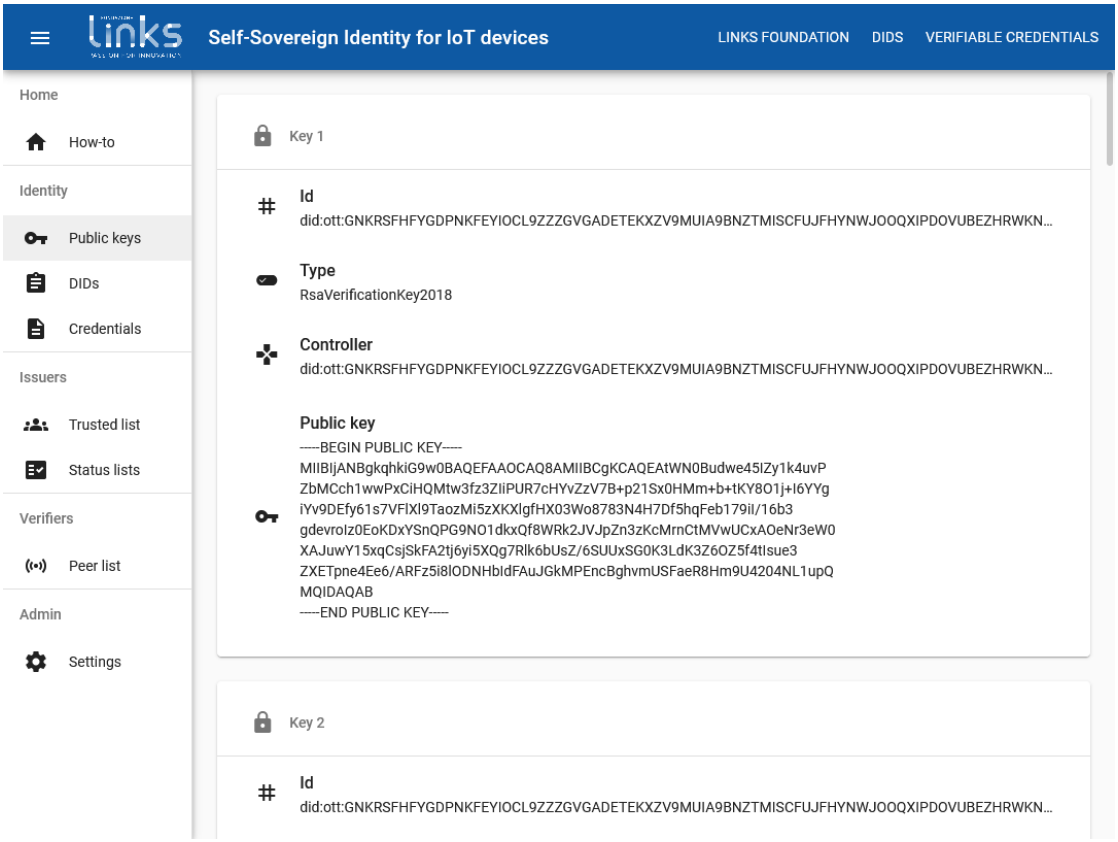


Figura 5.14: Pagina `/device/:owner/keys`.

Capitolo 6

Risultati

Nel seguente capitolo saranno illustrati i metodi seguiti per testare il framework e i risultati per i casi d'uso descritti nei capitoli precedenti. Saranno inoltre esposte le metodologie e i tool utilizzati.

6.1 Testing del framework

L'implementazione descritta nel Capitolo 4 è stata testata implementando le seguenti tipologie di test:

1. End-to-end: i peer vengono inizializzati, le richieste viaggiano sulla rete e i risultati vengono verificati nella stessa forma in cui l'applicazione è utilizzata in un caso reale.
2. Integration: i componenti vengono testati nel loro insieme, per verificare che le interazioni tra gli oggetti non producano errori e che i risultati delle funzioni siano coerenti con quelli attesi.
3. Unit: le funzioni degli oggetti vengono testate in maniera isolata dalle altre, non ci sono quindi interazioni tra le classi, per cui eventuali dipendenze vengono risolte introducendo oggetti mock.

Spesso sono necessarie delle operazioni di setup e di tear down affinché lo stato dell'applicazione permetta l'esecuzione test. Queste istruzioni infatti, sono utili per ripristinare lo stato dell'applicazione e per non ostacolare l'esecuzione dei test. Il codice è allora stato testato utilizzando il framework pytest [35], che semplifica il testing introducendo il concetto di fixture. Le fixture rappresentano il meccanismo con cui le istruzioni di setup e tear-down vengono gestite, e si basano sulla definizione di funzioni, decorate con l'annotazione `@pytest.fixture`, che specifica la modalità di esecuzione del codice. Listato 6.1 mostra un esempio di fixture in cui viene creato un Model che viene salvato in una Repository. Il codice di setup viene specificato prima della `yield`, che a differenza di una `return`, viene utilizzata per ritornare un risultato e riprendere l'esecuzione dopo la sospensione della funzione. Il codice di tear down invece, è quello compreso dopo la `yield`, che si occupa di eliminare dal database il Model creato.

```python
@pytest.fixture
def issuer_model(issuer_repo):
    iss_model = IssuerModel(_id="issuer_id", _did="issuer_did",
    ↪  _name="issuer_name")

    with issuer_repo as repo:
        repo.create(iss_model)
        repo.commit()

    yield iss_model

    with issuer_repo as repo:
        repo.delete(iss_model._id)
        repo.commit()
```

Listato 6.1: Esempio di fixture per la creazione di un nuovo `IssuerModel` (file `conftest.py`).

Una fixture può essere invocata specificando il suo nome nella signature del test case (Listato 6.2). Mediante poi l'argomento `scope` dell'annotazione `@pytest.fixture`, si può determinare l'esecuzione del codice, cioè *ogni quando* le istruzioni debbano essere ripetute: per ogni caso di test, per ogni modulo o per tutta la test suite, cioè una singola volta. Questo comportamento si rivela particolarmente utile per non appesantire l'esecuzione dei test, poiché evita che gli oggetti delle fixture siano continuamente ricreati senza motivo, risparmiando così risorse per l'avvio.

```python
def test_create_status_list(
    self,
    issuer_model: IssuerModel,
    ...
):
    issuer_did = issuer_model._did
    ...
```

Listato 6.2: Invocazione di una pytest fixture in un caso di test.

6.1.1 Funzioni mock

Talvolta, per alcuni casi di test occorre *simulare* determinati tipi di operazione, come chiamate di rete oppure accessi a database. Pytest offre quindi un particolare tipo di fixture denominata monkeypatch, che permette di ridefinire attributi, classi e funzioni. Tramite quindi un sistema di patching, si possono introdurre oggetti mock, che vanno

a sostituire quelle funzionalità non disponibili durante la fase di testing, e per attivare comportamenti altresì difficili da simulare mediante codice di setup.

Consideriamo quindi un primo esempio, che mette in evidenza come le funzioni mock semplificano la riproduzione di condizioni non previste.

Listato 6.3 mostra un caso di test, il cui obiettivo è quello di accertare che nella circostanza in cui il Tangle non trovi il DID document richiesto, il client che fa la richiesta lanci un'eccezione del tipo `RpcException`. Per simulare la condizione descritta, sarebbe sufficiente chiedere al Tangle di risolvere un DID document non esistente, ma ciò non garantirebbe la riproducibilità del caso di test. La soluzione migliore allora, è quella di essere certi che il metodo del Tangle lanci *sempre* un'eccezione del tipo `ModelNotFound⌋Exception`, in modo tale da verificare che la risposta attesa (`RpcException`) sia coerente con l'errore riscontrato dal Tangle, *a prescindere* dal DID document richiesto. Utilizzando quindi la fixture monkeypatch, occorre semplicemente sostituire la chiamata a funzione incaricata della risoluzione del DID document con una fittizia, che lanci l'eccezione prevista `ModelNotFoundException`. A causa del sistema di import del Python [42], è necessario applicare il patching dove il metodo viene *invocato* e non dove è *definito*, andando quindi ad agire direttamente nei sorgenti della libreria utilizzata. `monkeypatch.setattr` punta quindi alla funzione `call` all'interno del modulo `jsonrpcserver.async_dispatcher`, invocata per il dispatching delle richieste. L'effetto della `setattr` è quello di sostituire il metodo reale con la funzione mock `raise_not_found_exception`, che solleva un'eccezione del tipo `Mode⌋lNotFoundException` sul Tangle, ogni volta che il client richiede la resolve. Così facendo, è sufficiente verificare che la `resolve_document` della classe `DidMethodHandler` rilanci una `RpcException`, che rappresenta il tipo di eccezione attesa per la corretta riuscita del test.

```python
async def raise_not_found_exception(*args, **kwargs):
    raise ModelNotFoundException()

async def raise_runtime_error(*args, **kwargs):
    raise RuntimeError()

@pytest.mark.asyncio
async def test_resolve_doc_exceptions(
    ott_rpc_client: DidMethodHandler,
    did_document_model: DidDocumentModel,
    did_document_dto: DidDocumentDTO,
    monkeypatch: MonkeyPatch,
):

    monkeypatch.setattr(
        "jsonrpcserver.async_dispatcher.call",
        raise_not_found_exception,
        raising=True,
    )

    with pytest.raises(RpcException):
        _ = await ott_rpc_client.resolve_document(did=did_document_model._id)
```

Listato 6.3: Esempio di caso di test che fa uso di funzioni mock.

6.1.2 Classi mock

Un'altra utilità delle funzioni mock è quella di realizzare dei generatori di oggetti, la cui creazione richiederebbe una fase preliminare estremamente difficile da condensare in poche linee di codice. Definendo infatti delle fixture chiamate factory, è facile creare nuove istanze di oggetti che possono essere integrati oppure sostituiti con quelli veri.

```
class StreamReaderMock:
    def __init__(self):
        self._buffer: bytes = None
        self._event = asyncio.Event()

    async def read(self, length: int = 0) -> bytes:
        await self._event.wait()
        data = self._buffer
        self._buffer = None
        self._event.clear()
        return data

    async def readline(self) -> bytes:
        return "0".encode("utf-8")

    def set_data(self, data: bytes):
        self._buffer = data


class StreamWriterMock:
    def __init__(self, reader: StreamReaderMock):
        self._reader = reader
        self._buffer: bytes = None
        self._event = self._reader._event

    def write(self, data: bytes):
        self._buffer = data

    def write_eof(self):
        pass

    async def drain(self):
        self._reader.set_data(data=self._buffer)
        self._buffer = None
        self._event.set()
```

Listato 6.4: Implementazione di classi mock (file `test_handshake.py`).

Per eliminare la dipendenza con la rete, è necessario creare degli oggetti mock che vadano a sostituire le chiamate reali. Il processo di autenticazione tra due peer infatti, richiede che ci sia un nodo che abbia avviato un server, ed un altro che apra una nuova connessione. Nel caso reale, le comunicazioni di rete avvengono tramite le classi offerte dal modulo Python asyncio, `StreamReader` per le letture, e `StreamWriter` per le scritture, le cui istanze vengono generate dai metodi `open_connection` per le richieste lato client, e da `start_server` per le richieste lato server. Gli oggetti di questo tipo dipendono internamente da variabili difficili da riprodurre e di cui si sconsiglia la creazione tramite il costruttore [40]. La soluzione migliore allora, è quella di creare delle classi mock che ne simulino il comportamento.

Listato 6.4 mostra l'implementazione di due classi mock:

- `StreamReaderMock`: possiede un buffer di dati da cui è in grado di estrarre un contenuto. Rimane in attesa fino a a che il `StreamWriterMock` scrive dei dati. Il buffer viene svuotato e i dati vengono letti.
- `StreamWriterMock`: possiede un riferimento alla controparte reader. Inserisce i dati nel buffer del `StreamReaderMock` e lo libera dall'attesa.

Queste classi quindi, andranno a sostituire i reali oggetti Stream, fornendo un meccanismo di comunicazione basato sugli eventi e non sulle funzionalità di rete.

```
@pytest.fixture(scope="module")
def did_factory(
    mm: ModelMapper,
    did_document_repo: DidDocumentRepository,
    ...
):
    def _did_factory(owner: str, did: str, services: List[ServiceModel]):
        doc_dto = ...
        rsa_pub_model_authentication = RsaPublicCryptoMaterialModel(...)
        rsa_pub_model_assertion = RsaPublicCryptoMaterialModel(...)
        doc_model = DidDocumentModel(...)
        with did_document_repo:
            did_document_repo.create(doc_model)
            did_document_repo.commit()
        return doc_dto

    return _did_factory

@pytest.fixture
def reader_factory():
    def _reader_factory():
```

```
22          return StreamReaderMock()
23
24      yield _reader_factory
25
26
27  @pytest.fixture
28  def writer_factory():
29      def _writer_factory(reader: StreamReaderMock):
30          return StreamWriterMock(reader=reader)
31
32      yield _writer_factory
```

Listato 6.5: Implementazione di fixture factory (file `test_handshake.py`).

6.1.3 Esempio

Il test case mostrato nel Listato 6.6, rappresenta un esempio di come il processo di testing sia semplificato grazie alle dipendenze esplicite che intercorrono tra le classi del framework. Il `test_handshake_eax_mode` ha come obiettivo quello di verificare che l'handshake tra due peer sia corretto, utilizzando una `CipherMode` (Figura 3.9) di tipo EAX.

All'apertura di una nuova connessione, i nodi client creano due istanze di `StreamReader` e `StreamWriter` di cui si servono per la trasmissione dei dati. Le fixture `_factory` (Listato 6.5) rappresentano dei generatori di oggetti `Stream*Mock`, che realizzano una comunicazione di rete simulata. Durante il processo di autenticazione quindi, i peer eseguono l'handshake facendo viaggiare i dati attraverso i buffer degli oggetti mock creati dalle fixture, eseguendo però le stesse istruzioni degli oggetti reali. Le stesse considerazioni valgono anche per la classe `ConnectionContext`, tramite cui avviene la trasmissione dei dati cifrati, che non ha bisogno di essere ricreata completamente, poiché durante l'esecuzione dei test case fa uso di oggetti mock.

Facciamo inoltre notare come, affinché i peer possano completare i processi di autenticazione, debbano essere in possesso di una coppia di chiavi a crittografia pubblica. Questo requisito viene soddisfatto dalla `did_factory`, che si occupa di creare DID, DID document e relativo materiale crittografico *su richiesta*.

Il caso di test si conclude quindi con gli `assert`, per verificare che i DID e i DID document scambiati durante l'handshake corrispondano a quelli creati durante il setup, e che il canale sicuro derivato dal processo di autenticazione consenta di scambiare correttamente i messaggi in maniera confidenziale.

```
1  @pytest.mark.asyncio
2  async def test_handshake_eax_mode(
3      did_factory,
4      reader_factory,
5      writer_factory,
```

```
    server_handshake_mode_eax: ServerHandshakeMode,
    client_handshake_mode_eax: ClientHandshakeMode,
    ...
):
    server_doc_dto: DidDocumentDTO = did_factory(...)
    client_doc_dto: DidDocumentDTO = did_factory(...)

    ...
    server_reader = reader_factory()
    client_reader = reader_factory()
    server_writer = writer_factory(reader=client_reader)
    client_writer = writer_factory(reader=server_reader)

    server_cor = server_handshake_mode_eax.execute_handshake(
        reader=server_reader, writer=server_writer, mat=server_mat
    )

    client_cor = client_handshake_mode_eax.execute_handshake(
        reader=client_reader, writer=client_writer, mat=client_mat
    )

    res_list = await asyncio.gather(server_cor, client_cor)

    server_conn = res_list[0]
    client_conn = res_list[1]

    assert server_conn._did == server_did and server_conn._auth_doc ==
    ↪  client_doc_dto
    assert client_conn._did == client_did and client_conn._auth_doc ==
    ↪  server_doc_dto

    client_data = json.dumps(presentation_json)
    await client_conn.send(client_data)
    server_data = await server_conn.receive()
    assert client_data == server_data
    ...
```

Listato 6.6: Codice di test per il processo di autenticazione tra due peer in modalità EAX.

6.2 Profiling

Nell'ingegneria del software, per profiling si intende una forma di analisi, che ha come obiettivo quello fornire una misurazione delle prestazioni di un programma e della complessità delle sue istruzioni. Le informazioni ricavate dal profiling saranno poi utili per guidare il processo di ottimizzazione del software.

	Peer Holder	Peer Relay	IaaS
Issuer	PC	PC	PC
Verifier	PC	PC	PC
Relay	PC	i.MX6	PC
Holder	i.MX6	PC	i.MX6*

Tabella 6.1: Setup per tipo di nodo e test. Nel caso IaaS, la board è intesa come servizio per gli Holder.

La Tabella 6.1 illustra il setup degli esperimenti in laboratorio, eseguiti utilizzando un PC di architettura moderna e la board introdotta nella sezione 5.2. Gli esperimenti sono composti da diversi casi d'uso, che simulano degli utilizzi reali, ognuno indicato nella Tabella 6.2 come:

Peer Holder
Durante la fase di boot avviene la creazione del materiale crittografico e del relativo DID document, insieme al salvataggio dei modelli sul Tangle. L'Holder (i.MX6) richiede due credenziali all'Issuer, per l'accesso al servizio del Verifier e per l'inoltro tramite il Relay, e infine richiede una revoca. Vengono create tre Verifiable Presentation, una per ogni richiesta.

Peer Relay
Al boot vengono creati due DID, uno per le operazioni da Holder e l'altro per quelle da Verifier. Il Relay (i.MX6) richiede una credenziale che sarà utilizzata per completare l'inoltro al Verifier. Ricevuta una richiesta, crea una Verifiable Presentation e accede al servizio del Verifier, completando l'inoltro facendo pervenire la risposta all'Holder.

IaaS
Il servizio remoto viene avviato per servire un Holder, che farà tutti i tipi di richiesta descritti in precedenza, e in aggiunta richiederà la lista delle proprie Verifiable Credential, i propri DID document e il proprio materiale crittografico pubblico (chiavi RSA).

6.2.1 Profiling della memoria

La misurazione della quantità di memoria utilizzata dal codice in esecuzione su un dispositivo IoT è un'operazione critica, data la mancanza di risorse richieste dalle procedure aggiuntive per il profiling. Sebbene esistano tool già pronti per le misurazioni della memoria dinamica, tramite cui è sufficiente eseguire il codice lanciando gli script forniti, l'overhead introdotto per la generazione dei report e per la raccolta dei dati porta il dispositivo a

	Peer Holder	Peer Relay	IaaS
Creazione DID document	x1	x2	x2
Rilascio di credenziale	x2	x1	x1
Creazione Verifiable Presentation	x3	x1	x4
Accesso ad un servizio	x1	x1	x1
Inoltro di richiesta	x1	x1	x1
Revoca di credenziale	x1	-	x2

Tabella 6.2: Casi d'uso per ogni test.

fallire l'esecuzione, per via delle poche risorse a disposizione. La soluzione ottimale è stata quindi quella di sviluppare del codice da eseguire in background, in un processo separato, in grado di effettuare le misurazioni a cadenza periodica. Gli script vengono quindi eseguiti più volte, per registrare i picchi e raccogliere i dati su cui vengono calcolate le medie.

Il profiling della memoria si compone quindi di:

- Avvio di uno script che crea un sottoprocesso per far partire la funzione main su cui è definito il caso d'uso.
- Avvio della funzione main.
- Misurazione dell'utilizzo della RAM in background.

Così facendo si ottiene una misurazione priva di overhead.

La funzione `collect_mem_usage`, mostrata nel Listato 6.7, ha il compito di collezionare continuamente in background la memoria consumata dal processo identificato da `pid`, facendo uso della libreria psutil [31], che offre una suite di classi Python per l'acquisizione delle informazioni relative ai processi in esecuzione. La classe `Process` implementa i metodi per il monitoraggio dei processi, tra cui `memory_info`, che fornisce un'interfaccia semplificata verso le system call, compatibile con tutti i sistemi operativi. All'invocazione della `collect_mem_usage`, viene eseguito un ciclo infinito, in cui l'oggetto creato tramite la `memory_info` legge la quantità di memoria corrente consumata dal processo, rappresentato dalla variabile `p`. A cadenza regolare, la misurazione viene salvata in un array, insieme al picco attualmente registrato.

```
1 async def collect_mem_usage(pid: int, step: int) -> Tuple[int, int]:
2     p = psutil.Process(pid)
3     mem_usage = []
4     peak = 0
```

```
    avg = 0

    try:
        while True:
            current_usage = p.memory_info().rss
            if peak < current_usage:
                peak = current_usage
            mem_usage.append(p.memory_info().rss)
            await asyncio.sleep(step)
    except:  # noqa
        pass

    print("\nCollected mem usage (bytes):\n")
    print(mem_usage)
    print(f"\nPeak usage {peak/1024/1024} MB ({peak} bytes)")
    if len(mem_usage) > 0:
        avg = mean(mem_usage)
    print(f"Avg usage {avg/1024/1024} MB ({avg} bytes)\n")

    return peak, avg
```

Listato 6.7: Funzione per la misurazione del resident-set di un processo.

Listato 6.8 mostra un esempio di script utilizzato per avviare in background il profiling. La creazione del sottoprocesso avviene mediante la `create_subprocess_exec` del modulo asyncio, che si serve di un oggetto `Process` da cui estrae il pid necessario ad identificare il processo da misurare. La `communicate` della classe `Process` invece, avvia un'attesa bloccante, che si arresta nel momento in cui il processo termina. Il pid estratto allora, serve alla funzione `collect_mem_usage` su cui avviene l'attesa, per costruire un oggetto `psutil.Process`. Al termine dell'esecuzione della funzione main, cioè quella che avvia il sottoprocesso da misurare, l'oggetto `psutil.Process` perde il riferimento al processo identificato da `pid`, causando quindi un'eccezione del tipo `psutil.NoSuchProcess`. In questo modo, il ciclo infinito viene arrestato, e tutti i dati fino a quel momento raccolti vengono mandati in output. `asyncio.gather` infine, salva i risultati della `collect_mem_usage` e della `communicate`, su cui l'attesa era stata avviata *contemporaneamente*.

```
async def holder_track_mem(rpc_ip: str, ...):

    N = num_exec
    peak = 0
    avg = 0
    mem_usage = []
    ...

    for n in range(N):
```

```
10          try:
11              print(f"run n. {n+1}")
12              p = await asyncio.create_subprocess_exec(
13                  "python3",
14                  "src/holder_main.py",
15                  ... # altri parametri
16              )
17
18              res_list = await asyncio.gather(
19                  collect_mem_usage(p.pid, 0.05), p.communicate(),
                    ↪  return_exceptions=True
20              )
21
22              current_peak, current_avg = res_list[0]
23
24              mem_usage.append(current_avg)
25              if peak < current_peak:
26                  peak = current_peak
27          except:  # noqa
28              traceback.print_exc()
29              break
30
31      print(f"\nMaximum peak {peak/1024/1024} MB ({peak} bytes)")
32      if len(mem_usage) > 0:
33          avg = statistics.mean(mem_usage)
34      print(f"Avg of avgs {avg/1024/1024} MB ({avg} bytes)\n")
```

Listato 6.8: Codice per il profiling della memoria di un Holder (file `holder_exec_main.py`).

Per quanto invece riguarda il servizio remoto (sezione 5.3), le misurazioni del tempo in cui le richieste da parte degli Holder vengono servite sono registrate da funzioni middleware. Con il termine middleware, indichiamo quelle porzioni di codice eseguite da FastAPI prima e dopo l'elaborazione di una richiesta. Definendo quindi istruzioni che fanno partire un timer, prima che l'esecuzione sia lasciata alla funzione incaricata di servire la richiesta, è possibile misurare il tempo trascorso dalla ricezione fino all'invio della risposta. Un middleware simile è offerto dalla libreria FastAPI-utils [16], che include un Timing Middleware per valutare i tempi di risposta dei servizi implementati con FastAPI.

6.2.2 Utilizzo memoria

La Figura 6.1 illustra i risultati ottenuti, e riporta i valori verso cui l'utilizzo converge. Effettuando più misurazioni, si è osservato che i risultati mostrati subiscono variazioni di pochi MB, e che quindi possiamo considerare irrilevanti.

Dall'analisi della memoria heap, emerge che la maggior parte dello spazio viene utilizzato dall'importazione delle librerie esterne necessarie. La verifica è stata fatta con guppy3 [17], un tool che consente di ispezionare la memoria heap durante l'esecuzione di software Python.

Considerando invece solo il codice dei casi d'uso, escludendo quindi quello necessario a caricare i moduli esterni, l'utilizzo è di poco superiore ad 1 MB.

Per ultimo, dopo aver eseguito il test Peer Holder un numero di volte nell'ordine delle decine di migliaia, si è rilevato che un database contenente circa 20000 coppie di chiavi RSA e 10000 Verifiable Credential, arriva ad occupare un dimensione su disco di poco superiore ai 100 MB.

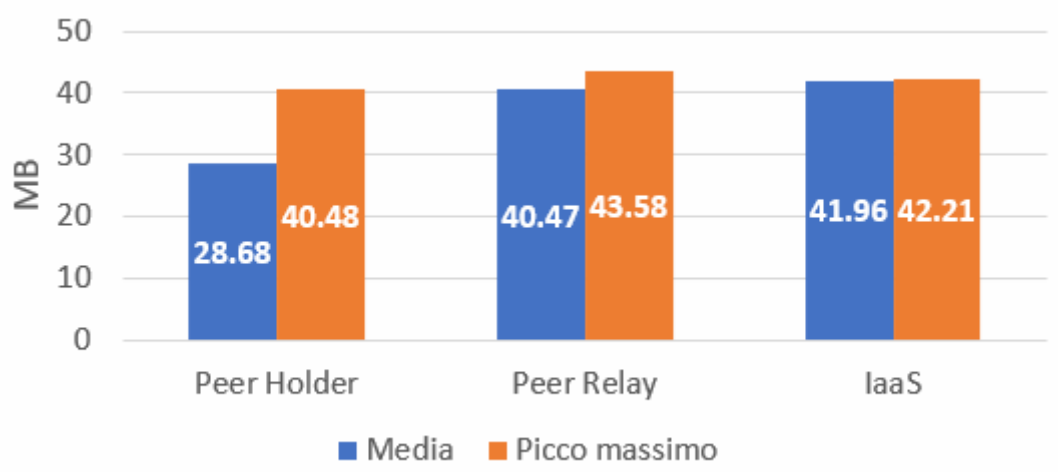


Figura 6.1: Utilizzo della memoria dinamica per caso d'uso.

6.2.3 Profiling CPU

La tipologia di profiling adottata per la misurazione dei tempi di esecuzione appartiene alla categoria dei profiler deterministici. In generale, un profiling di tipo deterministico fa uso di hook, su cui vengono registrate delle callback, che reagiscono al verificarsi di alcuni eventi scatenati dal sistema operativo, che però in Python, essendo un linguaggio interpretato, sono implementati via software [60]. I profiler deterministici quindi, effettuano una misurazione tutte le volte che l'evento su cui sono in ascolto viene scatenato. Al contrario, i profiler statistici basano le loro misurazioni su sampling a intervalli regolari. I sample sono rappresentati dagli stack di chiamate (snapshot), su cui viene fatta un'analisi statistica per dedurre il tempo speso dalle funzioni da misurare.

I profiler offerti dalla libreria standard del Python presentano alcune limitazioni, dovute all'overhead introdotto dall'invocazione degli hook e all'accuratezza dei tempi misurati [59]. La limitazione più grande però, è dovuta al fatto che in presenza di programmi multithread o asincroni, i risultati registrati non sono coerenti con il tempo effettivo speso durante le attese [64]. Il problema è accentuato ulteriormente dai profiler di tipo statistico, poiché dagli snapshot risulta estremamente complicato dedurre i tempi di attesa tra le funzioni asincrone. La scelta quindi è stata quella di adottare yappi [65], un profiler deterministico capace di misurare correttamente l'esecuzione di codice asincrono.

Il profiling dell'applicazione, viene avviato includendo il codice da misurare in un context manager, tramite il costrutto `with yappi.run()`, che avvia un event loop gestito da yappi, come mostrato nel Listato 6.9. Al termine dell'esecuzione viene creato un'oggetto `YFuncStats`, contenente tutti risultati raccolti. Per raggruppare i tempi per modulo allora, viene utilizzato un filtro (`yappi.module_matches()`) per generare una tabella (`print_all`) contenente solo i dati di interesse.

Facciamo notare infine, che il clock del profiling, cioè il tipo di misurazione, è stato impostato su `"WALL"`, che consente di misurare il tempo effettivo che intercorre tra l'inizio e la fine del programma, a differenza del clock `"CPU"`, che indica invece la quantità di tempo impiegato dalla CPU per completare le istruzioni.

```
loop = asyncio.get_event_loop()

async def _main():
    def __main():
        return main(
            tangle_address=tangle_address,
            tangle_port=tangle_port,
            ...
        )

if p is True:
    yappi.set_clock_type("WALL")
    with yappi.run():
        try:
            loop.run_until_complete(_main())
        except (Exception, KeyboardInterrupt, asyncio.CancelledError):
            traceback.print_exc()

    import service
    import repository
    import network
    import handler

    yappi.get_func_stats(
        filter_callback=lambda x: yappi.module_matches(
            x,
            [
                handler.authentication_handler,
                handler.authorization_handler,
                handler.did_method_handler,
                handler.exception_handler,
                handler.signature_verification_strategy,
                handler.verification_handler,
                network.bootstrap,
                ...
            ],
```

```
37          )
38      ).sort("ttot").print_all()
39
40  else:
41      loop.run_until_complete(_main())
```

Listato 6.9: Codice per il profiling dei tempi di esecuzione di un Holder.

6.2.4 Tempi di esecuzione

Dalle rilevazioni fatte sulla board risulta che le operazioni più costose in termini di risorse sono quelle relative alla creazione di chiavi RSA (Figura 6.2). Siccome quest'operazione è compresa due volte durante la creazione di un DID document, risulta essere l'operazione più lunga. Possiamo affermare che le funzioni eseguite durante il caso Peer Holder vengono completate in tempi accettabili, in relazione alle specifiche tecniche della board. Infatti, la creazione dei DID document (con relativo materiale crittografico) non è un'azione ricorrente, nel senso che questa operazione non è richiesta per ogni connessione. Riportiamo infine, che i tempi registrati nelle tabelle sono quelli relativi ad avvii successivi, cioè con un database già inizializzato.

Se consideriamo i tempi misurati su PC nel caso Peer Relay, vediamo che i tempi di risposta sono relativamente bassi quando il Relay è installato sulla board di riferimento (Figura 6.3). Le uniche differenze significative infatti, risiedono nel tempo speso per le operazioni crittografiche.

Mentre invece, per quanto riguarda il caso in cui il servizio remoto è installato sulla board, il tempo richiesto per soddisfare le richieste si attesta al di sotto del secondo (Figura 6.4). Considerando il tempo richiesto per la generazione delle chiavi, si potrebbe optare per introdurre un limite al numero di Holder da servire, e quindi evitare di saturare le risorse del dispositivo, che porterebbero inevitabilmente al blocco del servizio.

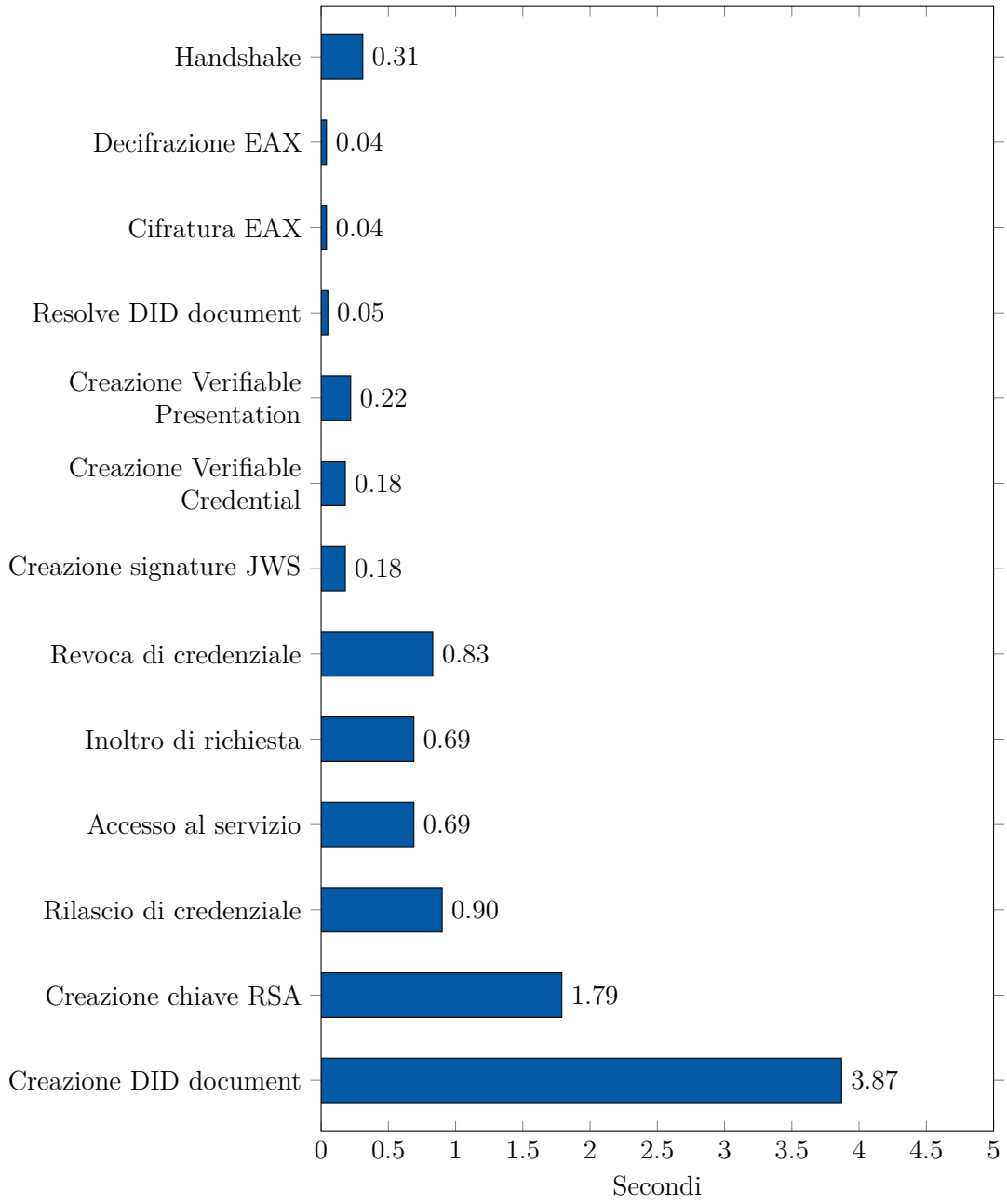


Figura 6.2: Peer Holder, tempi misurati su board i.MX6.

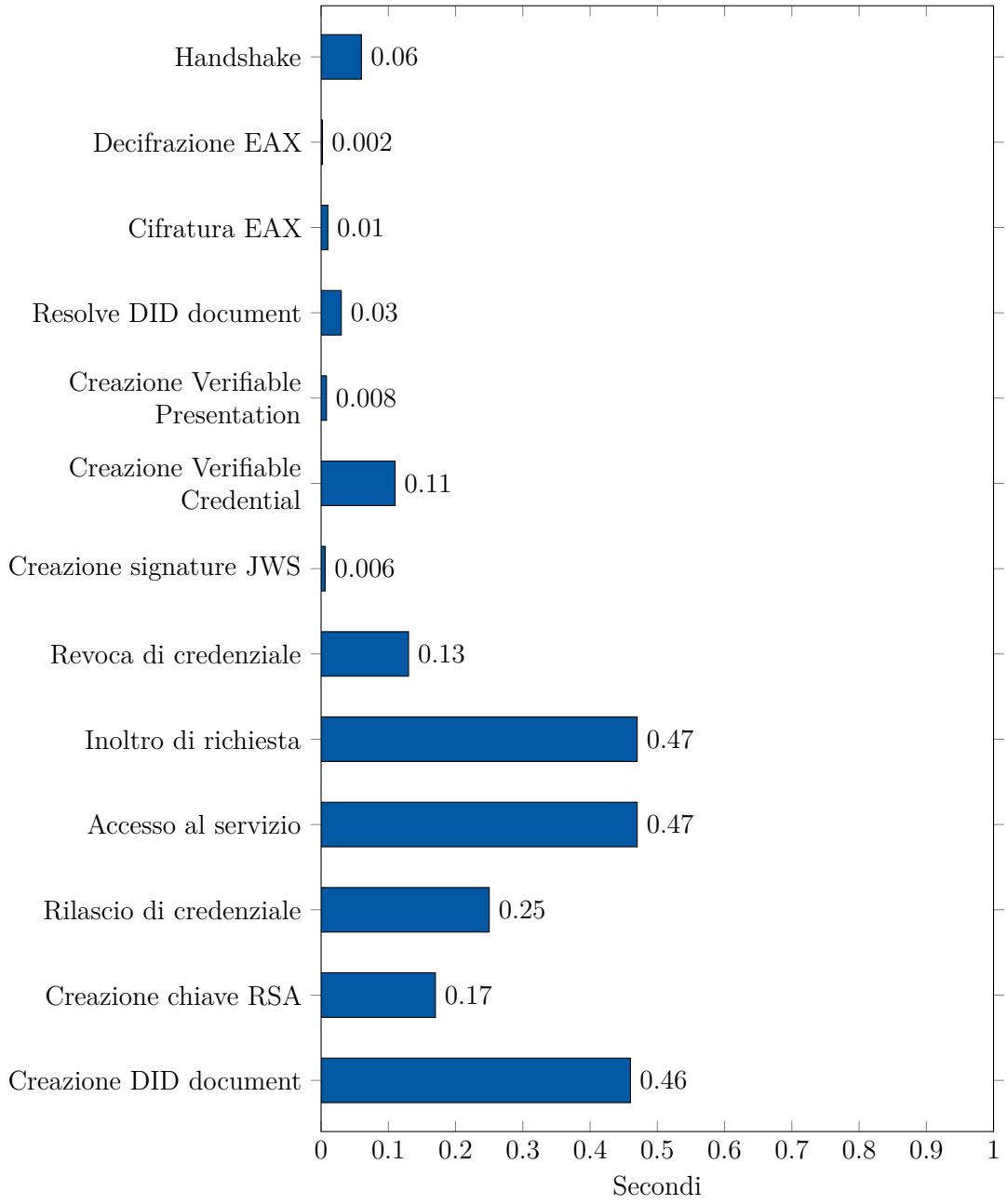


Figura 6.3: Peer Relay, tempi misurati su PC.

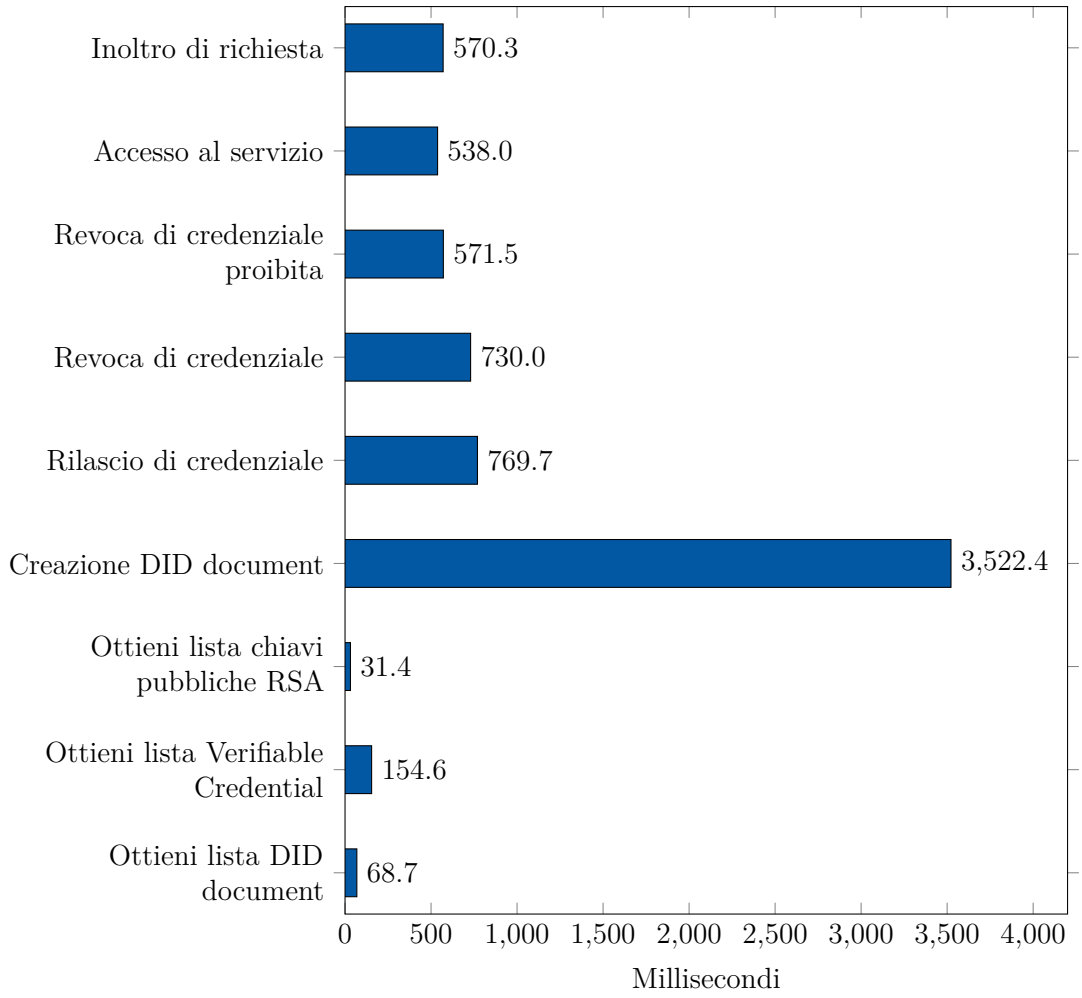


Figura 6.4: IaaS, tempi misurati su i.MX6.

Conclusioni e sviluppi futuri

In questo lavoro sono stati applicati i concetti di identità digitale ai dispositivi Internet of Things, attraverso lo sviluppo di un framework basato sulla Self-Sovereign Identity. A partire da un proof of concept realizzato dal team di Cybersecurity della Fondazione LINKS, il framework è stato ridefinito, implementato e testato su una board di architettura i.MX6, su cui è stata installata una distribuzione Linux custom compilata mediante la toolchain del progetto Yocto. Il percorso si conclude quindi con lo sviluppo di API REST che servono un applicativo a fini dimostrativi, che espone le funzionalità del framework attraverso un'interfaccia web interattiva.

L'implementazione proposta in questa tesi rappresenta un prototipo e non un prodotto finale, per cui è richiesta un'approfondita analisi di sicurezza.

I risultati ottenuti mettono in evidenza delle limitazioni dovute alle risorse richieste. Il passo successivo è quindi quello di migrare verso tecnologie a più alte prestazioni, che fanno uso di linguaggi compilati come il C/C++. C'è però da considerare lo sforzo richiesto per realizzare un framework simile, senza il supporto dei tool e delle librerie adottate in questa implementazione.

Lo scenario dell'Identity-as-a-Service richiede uno studio più approfondito, sia per valutare potenziali rischi dati dall'accesso al servizio, sia per l'architettura, che potrebbe introdurre problemi di centralizzazione e di scalabilità. Allo stato attuale il servizio offre funzionalità remote solo ai nodi di tipo Holder. Un ulteriore sviluppo è quello di integrare anche le funzioni degli altri tipi di nodo, in particolare dei Verifier, di cui non sono stati approfonditi i servizi erogati. Questa infatti, è limitata ad operazioni semplici, accessibili semplicemente presentando credenziali compatibili con il tipo di azione richiesta.

L'implementazione attuale non include le funzionalità più avanzate proposte dagli standard, quali dispute, refresh e delega di Verifiable Credential, e credenziali ad utilizzo singolo, ma soprattutto di un DID Method reale. Oltre a quelle citate, vi è un ulteriore meccanismo che aggiunge alle Verifiable Credential una proprietà aggiuntiva: il Blinded Link Secret. Un Holder infatti, può dimostrare ad un Verifier che le credenziali in suo possesso siano *veramente* state rilasciate a lui, poiché egli è l'unico a conoscere un segreto firmato dall'Issuer. Il framework va quindi esteso con i meccanismi di Blinded Link Secret e di proof of knowledge of a signature, introdotta nella sottosezione 2.3.4, così da integrare nel framework protocolli di sicurezza in zero-knowledge.

Uno dei problemi maggiori è però dovuto al processo di autenticazione, implementato come semplice protocollo di challenge-response. L'handshake è basato sullo scambio dei DID da parte dei due peer in comunicazione, che purtroppo però rende il protocollo di

autenticazione vulnerabile ad attacchi Man-in-the-middle.

Da riconsiderare è anche la scelta fatta per la gestione della persistenza delle dati, come Verifiable Credential e materiale crittografico. Gli strumenti adottati, pur essendo efficienti, non garantiscono una protezione allo stato dell'arte, per quei tipi di attacco che mirano a violare i contenuti salvati sui dispositivi. È necessario quindi, adottare contromisure adeguate, ad esempio introducendo nello stack del framework dei Secure Element per proteggere i dati critici.

Appendice A

Setup, build e installazione

Le istruzioni presenti in questo capitolo descrivono il processo che parte dalla compilazione di un'immagine minimale utilizzando la toolchain di Yocto, fino all'installazione su una board Variscite i.MX DART-6UL. I comandi forniti sono da eseguire su una macchina host su cui vi è installata una distribuzione di Ubuntu 18.04/20.04.

Parte delle istruzioni è stata ricavata da [13] e [18].

A.1 Compilazione con Yocto

A.1.1 Installazione dei requisiti

```
apt-get install gawk wget git diffstat unzip texinfo gcc-multilib \
build-essential chrpath socat cpio python python3 python3-pip python3-pexpect \
xz-utils debianutils iputils-ping libsdl1.2-dev xterm
```

```
apt-get install autoconf libtool libglib2.0-dev libarchive-dev python-git \
sed cvs subversion coreutils texi2html docbook-utils python-pysqlite2 \
help2man make gcc g++ desktop-file-utils libgl1-mesa-dev libglu1-mesa-dev \
mercurial automake groff curl lzop asciidoc u-boot-tools dos2unix mtd-utils \
pv libncurses5 libncurses5-dev libncursesw5-dev libelf-dev zlib1g-dev bc rename
```

```
apt-get install sqlcipher libssl-dev libsqlcipher-dev
```

A.1.2 Download di Yocto Dunfell

Clone della repository con `repo` [48]:

```
mkdir ~/bin
curl https://commondatastorage.googleapis.com/git-repo-downloads/repo >
~/bin/repo
chmod a+x ~/bin/repo
export PATH=~/bin:$PATH
```

Creazione della directory di lavoro:

```
mkdir ~/var-fslc-yocto
cd ~/var-fslc-yocto

repo init -u https://github.com/varigit/variscite-bsp-platform.git -b dunfell
repo sync -j$(nproc)
```

A.1.3 Inizializzazione directory di build

Creazione della directory di build e inizializzazione degli script di BitBake, da eseguire *solo* la prima volta:

```
cd ~/var-fslc-yocto
MACHINE=imx6ul-var-dart DISTRO=fslc-framebuffer . setup-environment build_fb
```

Se la directory `build_fb` è già esistente, è sufficiente inizializzare l'ambiente:

```
cd ~/var-fslc-yocto
source setup-environment build_fb
```

DISTRO messe a disposizione da Variscite:

fslc-x11
: X11 senza supporto wayland.

fslc-framebuffer
: Back-end grafico Framebuffer. Non include nè X11 nè wayland.

fslc-wayland
: Wayland senza supporto X11.

fslc-xwayland
: Wayland con supporto X11.

È possibile utilizzare anche le DISTRO fornite dalla distribuzione di riferimento Poky.

A.1.4 Integrazione di livelli aggiuntivi

Clonare da [28] i sorgenti dei seguenti meta-livelli, e fare il checkout al branch dunfell:

- `meta-webserver`
- `meta-cloud-services`
- `meta-openstack`
- `meta-sca`

Integrare quindi i livelli clonati e `meta-iotsec` utilizzando il comando:

```
bitbake-layers add-layer /path/to/meta-level-directory
```

In caso di conflitti sulle versioni dei package, incrementare la priorità dei meta-livelli aggiunti, modificando la variabile `BBFILE_PRIORITY_` contenuta nei file di configurazione `meta-level/conf/layer.conf`, come segue:

```
BBFILE_PRIORITY_meta-level = "10"
```

Aprire con un editor il file `build_fb/conf/local.conf` e aggiungere la seguente istruzione per aggiungere i package necessari:

```
IMAGE_INSTALL_append = "\
    mtd-utils \
    mtd-utils-ubifs \
    udev-rules-imx \
    imx-lib \
    imx-kobs \
    bash \
    openssh \
    sqlite3 \
    valgrind \
    python \
    python3 \
    python3-bitarray \
    pip-jsonrpcclient \
    python3-jsonrpcserver \
    python3-jsonschema \
    python3-numpy \
    python3-pycryptodome \
    pip-pyld \
```

```
python3-pytest \
python3-pytest-asyncio \
pip-pytest-cov \
pip-rstr \
python3-sqlalchemy \
python3-sqlite3 \
python3-termcolor \
python3-pip \
python3-toml \
pip-iniconfig \
python3-coverage \
pip-pyinstrument \
pip-fastapi \
pip-pydantic \
pip-devtools \
pip-email-validator \
pip-python-dotenv \
pip-typing-extensions \
pip-starlette \
python3-setuptools-scm \
pip-anyio \
pip-sniffio \
python3-yappi \
python3-cryptography \
pip-pprofile \
sqlcipher \
pip-httpx \
pip-guppy3 \
pip-uvicorn \
python3-idna \
python3-rfc3986 \
python3-certifi \
pip-charset-normalizer \
pip-h11 \
pip-httpcore \
python3-click \
pip-asgiref \
python3-psutil \
git \
gcc \
binutils \
sntp \
ntpq \
ntpdc \
dhcp-server \
dhcp-client \
python3-dev \
vim"
```

A.1.5 Compilazione

Compilazione immagine minimale:

```
bitbake core-image-minimal
```

Altre immagini fornite da Variscite e dalla FSL community BSP:

fsl-image-gui
Immagine demo default di Variscite con GUI e senza supporto a Qt5. Le recipe dell'immagine funzionano con tutti i back-end per X11, Framebuffer, wayland e xwayland, il contenuto è inoltre ottimizzato per l'installazione su una memoria NAND flash a 512MB.

fsl-image-qt5
Estende la fsl-image-gui aggiungendo il support a Qt5.

fsl-image-machine-test
Immagine console-only che include i package gstreamer, più applicazioni per benchmark e test.

fsl-image-mfgtool-initramfs
Immagine piccola che può essere utilizzata con il Manufacturing Tool (mfg-tool) in produzione.

Il risultato della compilazione si troverà in `var-fslc-yocto/build_fb/tmp/deploy/images/imx6ul-var-dart`. Per una descrizione dei file consultare [19].

A.2 Installazione dell'immagine

Per installare l'immagine compilata occorre preparare una SD card, avviare la board dallo slot SD, e poi lanciare lo script di installazione fornito da Variscite.

A.2.1 Preparazione della SD card

Per le istruzioni riportate sono fatte le seguenti assunzioni:

user
Nome dell'utente in Ubuntu.

/dev/sdb1
Partizione di boot della SD card.

/dev/sdb2
Partizione contenente il root file system nella SD card.

/mnt/rootfs
Cartella di mount per la partizione sdb2.

var-fslc-yocto
Directory di lavoro.

build_fb
Directory di build.

core-image-minimal
Immagine compilata con Yocto.

Unmount della scheda SD e copia dei file compilati nella sottosezione A.1.3:

```
sudo umount /dev/sdb*

zcat var-fslc-yocto/build_fb/tmp/deploy/images/imx6ul-var-dart/core-image-minim
al-imx6ul-var-dart.wic.gz | sudo dd of=/dev/sdb bs=1M &&
sync
```

Mount della partizione contenente il root filesystem nella SD card:

```
mkdir /mnt/rootfs
sudo mount /dev/sdb2 /mnt/rootfs
```

Avviare il seguente script, che copierà sulla scheda SD tutti i file necessari per installare l'immagine nelle memorie di storage della board:

```
#!/bin/sh

export YOCTO_IMGS_PATH=/home/user/var-fslc-yocto/build_fb/tmp/deploy/images/imx
6ul-var-dart/

export P2_MOUNT_DIR=/mnt/rootfs

sudo mkdir -p ${P2_MOUNT_DIR}/opt/images/Yocto/

sudo cp ${YOCTO_IMGS_PATH}/zImage ${P2_MOUNT_DIR}/opt/images/Yocto/

sudo cp ${YOCTO_IMGS_PATH}/*.dtb ${P2_MOUNT_DIR}/opt/images/Yocto/

sudo cp ${YOCTO_IMGS_PATH}/SPL-nand ${P2_MOUNT_DIR}/opt/images/Yocto/

sudo cp ${YOCTO_IMGS_PATH}/u-boot.img-nand ${P2_MOUNT_DIR}/opt/images/Yocto/
```

```
sudo cp ${YOCTO_IMGS_PATH}/core-image-minimal-imx6ul-var-dart_128kbpeb.ubi
${P2_MOUNT_DIR}/opt/images/Yocto/rootfs_128kbpeb.ubi

sudo cp /home/user/var-fslc-yocto/sources/meta-variscite-fslc/scripts/var_mk_yo
cto_sdcard/variscite_scripts/mx6_install_yocto.sh
${P2_MOUNT_DIR}/usr/bin

sudo cp /home/user/var-fslc-yocto/sources/meta-variscite-fslc/scripts/var_mk_yo
cto_sdcard/variscite_scripts/echos.sh
${P2_MOUNT_DIR}/usr/bin

echo "COPIED"
```

Nel caso in cui lo spazio per la copia dei file non dovesse essere sufficiente, occorre ridimensionare la partizione `/dev/sdb2` (root file system). Si consiglia ad esempio di utilizzare fdisk.

Impostare i PIN della board a 00 per l'avvio da scheda SD:

Al boot, fare il login inserendo "root" come password, quindi lanciare lo script `mx6_install_yocto.sh` copiato precedentemente in `/usr/bin`.

Appendice B

Istruzioni per l'esecuzione

L'esecuzione del framework nel Capitolo 4 necessita dell'installazione di una distribuzione Python superiore o uguale alla 3.8. Per l'applicazione dimostrativa (sezione 5.4) invece, è necessario installare una versione di Node.js superiore o uguale alla 14.

B.1 Avvio del framework

Installazione dei requisiti:

```
pip3 install -r requirements.txt
```

Il Tangle deve essere avviato per primo, l'Holder per ultimo.
Per ogni script, la lista degli argomenti è visualizzabile specificando l'opzione `-h`.

B.1.1 Tangle

```
python3 src/tangle_main.py
```

`--ip`
: Indirizzo IP in ascolto per le richieste.

`--port`
: Porta in ascolto.

`--secret`
: Segreto con cui cifrare il database locale, se non specificato sarà in chiaro.

B.1.2 Issuer

```
python3 src/issuer_main.py
```

`--rpc-ip`
Indirizzo IP del Tangle.

`--rpc-port`
Porta del Tangle.

`--ip`
Indirizzo IP in ascolto.

`--issue-port`
Porta per il servizio di rilascio credenziali.

`--revoke-port`
Porta per il servizio di revoca credenziali.

`--pool-size`
Dimensione del pool di Issuer noti (trusted list).

`--issuer-index`
Indice del pool per selezionare l'Issuer da avviare.

`--secret`
Segreto con cui cifrare il database locale, se non specificato sarà in chiaro.

B.1.3 Verifier

```
python3 src/verifier_main.py
```

`--rpc-ip`
Indirizzo IP del Tangle.

`--rpc-port`
Porta del Tangle.

`--ip`
Indirizzo IP in ascolto.

`--starting-port`
Porta di partenza per i servizi del Verifier.

`--pool-size`
Dimensione del pool di Verifier noti.

`--verifier-index`
Indice del pool per selezionare il Verifier da avviare.

`--issuer-pool-size`
Dimensione dal pool degli Issuer noti (trusted list).

`--secret`
Segreto con cui cifrare il database locale, se non specificato sarà in chiaro.

B.1.4 Relay

```
python3 src/relay_main.py
```

`--rpc-ip`
Indirizzo IP del Tangle.

`--rpc-port`
Porta del Tangle.

`--ip`
Indirizzo IP in ascolto.

`--port`
Porta in ascolto.

`--issuer-pool-size`
Dimensione del pool di Issuer noti (trusted list).

`--verifier-pool-size`
Dimensione del pool di Verifier noti.

`--issuer-index`
Indice del pool per selezionare l'Issuer a cui inviare le richieste.

`--relay-index`
Indice del pool per selezionare il Verifier da avviare come Relay.

`--name`
Nome dell'Holder.

`--secret`
Segreto con cui cifrare il database locale, se non specificato sarà in chiaro.

B.1.5 Holder

```
python3 src/holder_main.py
```

`--rpc-ip`
Indirizzo IP del Tangle.

`--rpc-port`
Porta del Tangle.

`--issuer-pool-size`
Dimensione del pool di Issuer noti (trusted list).

`--verifier-pool-size`
Dimensione del pool di Verifier noti.

`--issuer-index`
Indice del pool per selezionare l'Issuer a cui inviare le richieste.

`--verifier-index`
Indice del pool per selezionare il Verifier a cui inviare le richieste.

`--relay-index`
Indice del pool (dei Verifier) per selezionare il Relay a cui inviare le richieste.

`--name`
Nome dell'Holder.

`--did`
DID già creato da utilizzare come Holder.

`--secret`
Segreto con cui cifrare il database locale, se non specificato sarà in chiaro.

`-p` Attiva il profiling della CPU.

`-m` Attiva il profiling della memoria.

B.1.6 Servizio remoto

```
python3 src/holder_app.py
```

`--rpc-ip`
Indirizzo IP del Tangle.

`--rpc-port`
: Porta del Tangle.

`--issuer-pool-size`
: Dimensione del pool di Issuer noti (trusted list).

`--verifier-pool-size`
: Dimensione del pool di Verifier noti.

`--secret`
: Segreto con cui cifrare il database locale, se non specificato sarà in chiaro.

`-m` Attiva il profiling della memoria.

B.2 Test suite

La cartella `test` contiene i file che definiscono i test suddivisi per tipologia. Per avviare l'intera test suite è sufficiente:

```
pytest
```

Mentre invece, per l'avvio di test case specifici occorre lanciare:

```
pytest test/<dir>/<test_*.py>::<class_name>::<test_name>
```

Se nel file `<test_*.py>` non è presente una classe di test, `<class_name>` può essere omesso.

B.3 Esecuzione del dimostrativo

Per eseguire il dimostrativo è necessario dapprima installare i package necessari e poi lanciare un server in locale con i comandi forniti da Angular. In alternativa, configurare il server come descritto in [23] e seguire le istruzioni della sottosezione B.3.3. Infine, avviare i nodi Tangle, Issuer, Verifier, Relay, e il server che espone le API REST (sottosezione B.1.6).

B.3.1 Installazione dei package

Posizionarsi nella directory root e lanciare:

```
npm install
```

Angular scaricherà tutte le dipendenze all'interno della cartella `node_modules`.

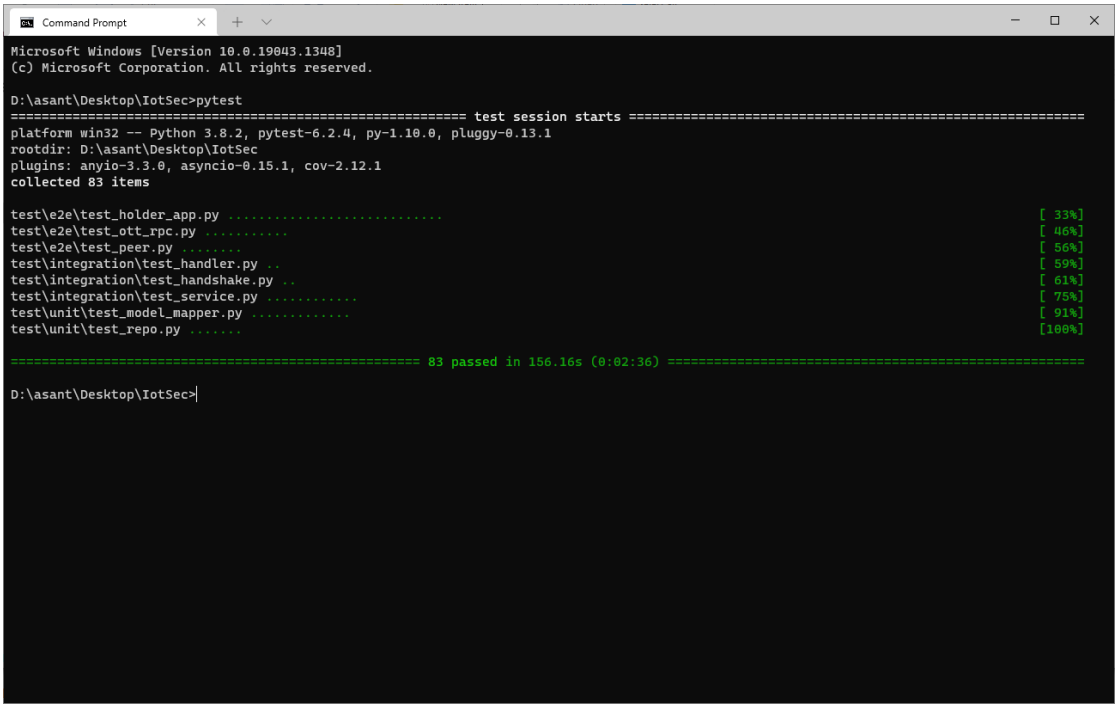


Figura B.1: Esecuzione della test suite completa.

B.3.2 Avvio server locale con Webpack

Il seguente comando fornito da Angular permette di lanciare un server in locale tramite Webpack [63], localizzato all'indirizzo `http://localhost:4200/`.

```
ng serve
```

B.3.3 Deploy su server nginx

Compilare l'applicazione con:

```
ng build
```

I file da copiare in nginx saranno contenuti in una sottocartella all'interno di `dist`. Utilizzare quindi la seguente configurazione di base:

```
server {
    # HTTPS
```

```
    # listen 443 ssl;
    # listen [::]:443 ssl;

    # ssl_certificate /etc/nginx/certificates/cert.crt;
    # ssl_certificate_key /etc/nginx/certificates/k.key;

    listen 80;
    listen [::]:80;

    server_name 127.0.0.1;

    root /var/www/dir_name;
    location / {
        try_files $uri $uri/ /index.html;
    }

    location /api {
        proxy_pass http://127.0.0.1:8000;
        rewrite /api/(.*) /$1 break;
        add_header 'Access-Control-Allow-Origin' '*' always;
    }
}
```

Bibliografia

[1] Joe Andrieu et al. *DID Method Rubric v1.0*. URL: https://w3c.github.io/did-rubric/. (accesso: 08.07.2021).

[2] *Angular documentation - FormControl API*. URL: https://angular.io/api/forms/FormControl. (accesso: 26.11.2021).

[3] *Angular homepage*. URL: https://angular.io/. (accesso: 26.11.2021).

[4] *Angular Material documentation - chips API reference*. URL: https://material.angular.io/components/chips/api. (accesso: 26.11.2021).

[5] *Angular Material documentation - progress-spinner API reference*. URL: https://material.angular.io/components/progress-spinner/api. (accesso: 26.11.2021).

[6] *Angular Material documentation - radio API reference*. URL: https://material.angular.io/components/radio/api. (accesso: 26.11.2021).

[7] *Angular Material documentation - slide-toggle API reference*. URL: https://material.angular.io/components/slide-toggle/api. (accesso: 26.11.2021).

[8] *Angular Material documentation - stepper API reference*. URL: https://material.angular.io/components/stepper/api. (accesso: 26.11.2021).

[9] Luigi Atzori, Antonio Iera e Giacomo Morabito. «The Internet of Things: A Survey». In: *Computer Networks* (ott. 2010), pp. 2787–2805. DOI: 10.1016/j.comnet.2010.05.010.

[10] *bitarray repository*. URL: https://github.com/ilanschnell/bitarray. (accesso: 05.11.2021).

[11] Jan Camenisch e Anna Lysyanskaya. «A Signature Scheme with Efficient Protocols». In: gen. 2002, pp. 268–289.

[12] Julien Danjou. *Serious Python: Black-Belt Advice on Deployment, Scalability, Testing, and More*. A cura di No Starch Press. 2018.

[13] *DART-6UL - Yocto Dunfell 3.1 based on FSL Community BSP 3.1 with 5.4.24_2.1.0 Linux release*. URL: https://variwiki.com/index.php?title=Yocto_Build_Release&release=RELEASE_DUNFELL_V1.1_DART-6UL. (accesso: 12.11.2021).

[14] Matthew Davie et al. *The Trust Over IP Stack*. URL: https://github.com/hyperledger/aries-rfcs/tree/master/concepts/0289-toip-stack. (accesso: 05.07.2021).

[15] *FastAPI documentation*. URL: https://fastapi.tiangolo.com/. (accesso: 20.11.2021).

[16] *FastAPI-utils repository*. URL: https://github.com/dmontagu/fastapi-utils. (accesso: 25.11.2021).

[17] *guppy3 - A Python Programming Environment and Heap analysis toolset*. URL: https://zhuyifei1999.github.io/guppy3/. (accesso: 06.11.2021).

[18] *Installing Yocto to the SOM's internal storage*. URL: https://variwiki.com/index.php?title=Yocto_NAND_Flash_Burning&release=RELEASE_DUNFELL_V1.1_DART-6UL. (accesso: 12.11.2021).

[19] *Installing Yocto to the SOM's internal storage - Build Results*. URL: https://variwiki.com/index.php?title=Yocto_Build_Release&release=RELEASE_DUNFELL_V1.1_DART-6UL#Build_Results. (accesso: 12.11.2021).

[20] Dave Longley e Manu Sporny. *Status List 2021: Privacy-preserving status information for Verifiable Credentials*. URL: https://w3c-ccg.github.io/vc-status-list-2021/. (accesso: 06.07.2021).

[21] Carsten Maple. «Security and privacy in the internet of things». In: *Journal of Cyber Policy* 2 (mag. 2017), pp. 155–184. DOI: 10.1080/23738871.2017.1366536.

[22] Arsalan Mosenia e Niraj K. Jha. «A Comprehensive Study of Security of Internet-of-Things». In: *IEEE Transactions on Emerging Topics in Computing* 5.4 (2017), pp. 586–602. DOI: 10.1109/TETC.2016.2606384.

[23] *nginx documentation*. URL: https://nginx.org/en/docs/. (accesso: 12.11.2021).

[24] *NumPy documentation*. URL: https://numpy.org/doc/stable/index.html. (accesso: 05.11.2021).

[25] *NumPy documentation - frombuffer API reference*. URL: https://numpy.org/doc/stable/reference/generated/numpy.frombuffer.html?highlight=frombuffer#numpy.frombuffer. (accesso: 05.11.2021).

[26] *NumPy documentation - uint32 API reference*. URL: https://numpy.org/doc/stable/reference/arrays.scalars.html?highlight=uint32#numpy.uint32. (accesso: 05.11.2021).

[27] Darrell O'Donnell, Mike Vesey e Drummond Reed. «Learn about the Trust Over IP (ToIP) stack». In: 2020. URL: https://ssimeetup.org/trust-over-ip-toip-stack-webinar-54/.

[28] *Open Embedded Layer Index - Dunfell branch*. URL: https://layers.openembedded.org/layerindex/branch/dunfell/layers/. (accesso: 05.12.2021).

[29] Dusty Phillips. *Python 3 Object-Oriented Programming: Build robust and maintainable software with object-oriented design patterns in Python 3.8, 3rd Edition*. A cura di Packt Publishing Limited. 2018.

[30] *pickle - Python object serialization*. URL: https://docs.python.org/3.8/library/pickle.html. (accesso: 06.07.2021).

[31] *psutil documentation*. URL: https://psutil.readthedocs.io/en/latest/. (accesso: 20.11.2021).

[32] *pyca/cryptography documentation*. URL: https://cryptography.io/en/latest/. (accesso: 03.11.2021).

[33] *PyCryptodome documentation*. URL: https://pycryptodome.readthedocs.io/en/latest/index.html. (accesso: 03.11.2021).

[34] *pydantic documentation*. URL: https://pydantic-docs.helpmanual.io/. (accesso: 20.11.2021).

[35] *pytest documentation*. URL: https://docs.pytest.org/en/6.2.x/. (accesso: 25.10.2021).

[36] *Python 3.8 documentation - asyncio - Asynchronous I/O*. URL: https://docs.python.org/3.8/library/asyncio.html. (accesso: 06.07.2021).

[37] *Python 3.8 documentation - asyncio - Streams*. URL: https://docs.python.org/3.8/library/asyncio-stream.html. (accesso: 04.11.2021).

[38] *Python 3.8 documentation - base64 - Base16, Base32, Base64, Base85 Data Encodings*. URL: https://docs.python.org/3.8/library/base64.html. (accesso: 05.11.2021).

[39] *Python 3.8 documentation - sqlite3 - DB-API 2.0 interface for SQLite databases*. URL: https://docs.python.org/3.8/library/sqlite3.html. (accesso: 06.07.2021).

[40] *Python 3.8 documentation - Streams - StreamReader class*. URL: https://docs.python.org/3.8/library/asyncio-stream.html#streamreader. (accesso: 20.10.2021).

[41] *Python 3.8 documentation - struct - Interpret bytes as packed binary data*. URL: https://docs.python.org/3.8/library/struct.html. (accesso: 05.11.2021).

[42] *Python 3.8 documentation - unittest.mock - mock object library - Where to patch*. URL: https://docs.python.org/3.8/library/unittest.mock.html#where-to-patch. (accesso: 20.10.2021).

[43] *Python 3.8 documentation - zlib - Compression compatible with gzip*. URL: https://docs.python.org/3.8/library/zlib.html. (accesso: 05.11.2021).

[44] Drummond Reed. «Decentralized Identifiers (DIDs): The Fundamental Building Block of Self-Sovereign Identity». In: 2018. URL: https://ssimeetup.org/decentralized-identifiers-did-fundamental-block-self-sovereign-identity-drummond-reed-webinar-2/.

[45] Drummond Reed. «The Story of Open SSI Standards». In: 2018. URL: https://ssimeetup.org/story-open-ssi-standards-drummond-reed-evernym-webinar-1/.

[46] Drummond Reed, Dave Longley e David Chadwick. *Verifiable Credentials Data Model 1.0*. URL: https://www.w3.org/TR/vc-data-model/. (accesso: 02.07.2021).

[47] Drummond Reed et al. *Decentralized Identifiers (DIDs) v1.0*. URL: https://www.w3.org/TR/did-core/. (accesso: 30.06.2021).

[48] *repo - The Multiple Git Repository Tool*. URL: https://gerrit.googlesource.com/git-repo/. (accesso: 12.11.2021).

[49] Rodrigo Roman, Jianying Zhou e Javier Lopez. «On the features and challenges of security and privacy in distributed Internet of things». In: *Computer Networks* 57 (lug. 2013), pp. 2266–2279. DOI: 10.1016/j.comnet.2012.12.018.

[50] Tyler Ruff. «Verifiable Credentials 101 for SSI». In: 2018. URL: https://ssimeetup.org/verifiable-credentials-101-ssi-tyler-ruff-webinar-11/.

[51] *RxJS - Overview*. URL: https://rxjs.dev/guide/overview. (accesso: 24.11.2021).

[52] Brett Slatkin. *Effective Python: 90 Specific Ways to Write Better Python*. A cura di Pearson Education (US). 2020.

[53] *SQLAlchemy 1.4 documentation*. URL: https://docs.sqlalchemy.org/en/14/intro.html. (accesso: 06.07.2021).

[54] *SQLAlchemy 1.4 documentation - Dialects*. URL: https://docs.sqlalchemy.org/en/14/dialects/index.html. (accesso: 06.07.2021).

[55] *SQLAlchemy 1.4 documentation - Performance*. URL: https://docs.sqlalchemy.org/en/14/faq/performance.html. (accesso: 06.07.2021).

[56] *SQLCipher Community Edition design documentation*. URL: https://www.zetetic.net/sqlcipher/design/. (accesso: 03.11.2021).

[57] *SQLCipher Performance and SQLCipherSpeed*. URL: https://www.zetetic.net/blog/2011/5/7/sqlcipher-performance-and-sqlcipherspeed.html. (accesso: 03.11.2021).

[58] Orie Steele e Manu Sporny. *DID Specification Registries*. URL: https://w3c.github.io/did-spec-registries/. (accesso: 08.07.2021).

[59] *The Python Profilers - Limitations*. URL: https://docs.python.org/3.8/library/profile.html#limitations. (accesso: 20.10.2021).

[60] *The Python Profilers - What Is Deterministic Profiling?* URL: https://docs.python.org/3.8/library/profile.html#what-is-deterministic-profiling. (accesso: 20.10.2021).

[61] Andrew Tobin e Drummond Reed. «The Inevitable Rise of Self-Sovereign Identity». In: (2017).

[62] *Variscite wiki - DART-6UL*. URL: https://variwiki.com/index.php?title=DART-6UL. (accesso: 23.10.2021).

[63] *Webpack documentation - Concepts*. URL: https://webpack.js.org/concepts/. (accesso: 12.11.2021).

[64] *yappi documentation - Profiling Coroutines*. URL: https://github.com/sumerc/yappi/blob/master/doc/coroutine-profiling.md. (accesso: 20.10.2021).

[65] *yappi repository*. URL: https://github.com/sumerc/yappi. (accesso: 20.10.2021).

[66] *Yocto Project documentation - Introducing the Yocto Project*. URL: https://docs.yoctoproject.org/3.1.10/overview-manual/overview-manual-yp-intro.html. (accesso: 05.10.2021).

[67] *Yocto Project documentation - Yocto Project Concepts*. URL: `https://docs.yoctoproject.org/3.1.10/overview-manual/overview-manual-concepts.html`. (accesso: 05.10.2021).

Elenco delle figure

2.1 Esempio di decentralized identifier (DID). Immagine tratta da [47]. 15
2.2 Architettura dettagliata dei Decentralized Identifier e relazioni tra le componenti. Immagine tratta da [47]. 18
2.3 Production e consumption di rappresentazioni diverse. Immagine tratta da [47]. 19
2.4 Attori nell'ecosistema delle Verifiable Credential. Immagine tratta da [46]. 20
2.5 Claim multipli che esprimono un'informazione contenuta in una Verifiable Credential. Immagine tratta da [46]. 20
2.6 Interazioni tra i diversi tipi di peer. Immagine tratta da [46]. 25
2.7 I due "triangoli della fiducia" per i tre diversi peer. Immagine tratta da [14]. 28
2.8 I livelli governativi e tecnologici del Trust over IP stack. Immagine tratta da [14]. 28

3.1 Moduli del framework. 32
3.2 Diagramma delle classi per un nodo Issuer. 33
3.3 Diagramma delle classi per le funzioni di verifica. 34
3.4 Diagramma delle classi per un nodo Verifier. 35
3.5 Diagramma delle classi per un nodo Relay. 36
3.6 Classi utilizzate da un Holder. 37
3.7 Classi che compongono il Tangle. 37
3.8 Diagramma delle classi che compongono gli endpoint e le componenti per l'autenticazione tra due peer. 39
3.9 Diagramma delle classi che permettono di definire la modalità di cifratura, decifrazione e di autenticazione. 40

4.1 Passi di challenge-response per il protocollo di autenticazione. 59
4.2 Status bitmap. Immagine tratta da [20]. 62

5.1 Struttura dei livelli indispensabili per una compilazione. Il Distro Layer (policy) e il BSP Layer (package per le board specifiche) vengono forniti dai produttori dell'architettura target. Immagine tratta da [67]. 74
5.2 Struttura delle directory di Poky. Il file `oe-init-build-env` contiene tutti gli script per Poky e BitBake e permette di generare una cartella preconfigurata per la compilazione. Immagine tratta da [67]. 78

5.3 Rappresentazione ad alto livello del workflow del sistema di build OpenEmbedded. Immagine tratta da [67]. 78
5.4 NXP i.MX DART-6UL. Immagine tratta da [62]. 79
5.5 Processo di compilazione con Yocto. 82
5.6 Pagina `/device/:owner/dids`. 98
5.7 Pagina `/device/:owner/credentials/use`. 102
5.8 Pagina `/device/:owner/credentials`. 105
5.9 Pagina `/device/:owner/credentials`. 109
5.10 Pagina `/device/:owner/credentials/:id/use`. 115
5.11 Diagramma marble dell'operatore `switchMap`. 116
5.12 Diagramma marble dell'operatore `mergeMap` (o `flatMap`). 120
5.13 Pagina `/device/:owner/credentials`. 122
5.14 Pagina `/device/:owner/keys`. 123

6.1 Utilizzo della memoria dinamica per caso d'uso. 135
6.2 Peer Holder, tempi misurati su board i.MX6. 138
6.3 Peer Relay, tempi misurati su PC. 139
6.4 IaaS, tempi misurati su i.MX6. 140

B.1 Esecuzione della test suite completa. 155

Elenco delle tabelle

1.1 Requisiti di sicurezza. 9

5.1 Specifiche della board NXP i.MX DART-6UL. 80
5.2 Lista delle route con componenti utilizzati. Le pagine che includono path comuni creano Component di entrambe le route, ad esempio `/device/:owner/dids` include gli stessi Component della pagina `/device/:owner`. . 94
5.3 Lista delle route con relative funzionalità. 99

6.1 Setup per tipo di nodo e test. Nel caso IaaS, la board è intesa come servizio per gli Holder. 131
6.2 Casi d'uso per ogni test. 132