UNIVERSITÀ DEGLI STUDI DI MODENA
E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche e
Matematiche

Corso di Laurea Magistrale in Informatica

Tesi di Laurea

ACRATE, set di librerie Rust per l'interoperabilità

Relatore:

Prof.ssa
Claudia Canali

Relatore:

Dott. Ric.
Francesco Faenza

Candidato:

Dott.
Stefano Pigozzi

Matr. 177698

Anno Accademico 2024-2025

Indice

1. Sinossi 5
2. Introduzione 6
3. Concetti preliminari 8
   3.1. Rust 8
   3.2. Servizi web 16
   3.3. cURL 18
   3.4. XML 19
   3.5. JSON 19
   3.6. PostgreSQL 19
   3.7. Docker 20
   3.8. Resource descriptor 23
   3.9. NodeInfo 24
   3.10. JSON-LD 25
   3.11. ActivityStreams 30
4. Lavoro svolto 33
   4.1. Selezione del linguaggio di programmazione 33
   4.2. Creazione del workspace 33
   4.3. Implementazione dei resource descriptor in Rust 34
   4.4. Creazione servizio web di generazione resource descriptor 41
   4.5. Implementazione di NodeInfo in Rust 54
   4.6. Definizione di astrazioni per interagire con JSON-LD 55
   4.7. Realizzazione di macro per implementare le astrazioni JSON-LD 80
   4.8. Generazione delle implementazioni dei tipi di ActivityStreams 88
   4.9. Selezione di una licenza 93
5. Conclusioni e sviluppi futuri 94
   5.1. Investigazione di un errore di compilazione inaspettato 95
   5.2. Pubblicazione come crate separate 95
   5.3. Aggiunta di funzionalità di documentazione delle implementazioni generate 96
   5.4. Generazione automatica di implementazioni da schemi JSON-LD 97
Bibliografia 108

1. Sinossi

Questa tesi descrive lo sviluppo di ACRATE, un insieme di crate libreria per il linguaggio di programmazione Rust che implementano standard di Internet per permettere e facilitare la realizzazione di applicazioni interoperabili ed eventualmente federate.

Figura A: L'icona del progetto, una scatola bianca su sfondo arancione.

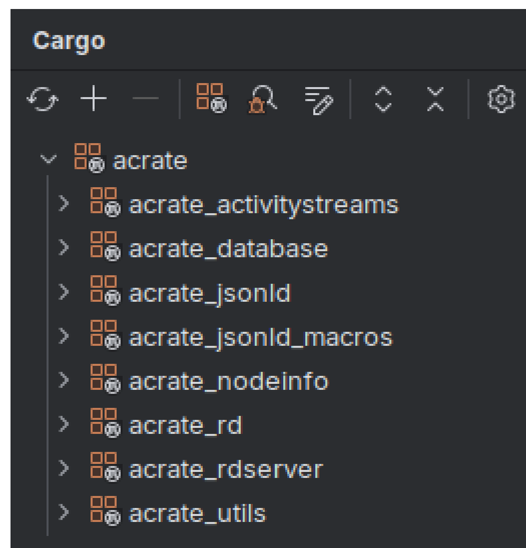

Figura B: Screenshot del pannello Cargo di IntelliJ IDEA, mostrante il workspace Rust ACRATE e le relative crate contenute in esso.

2. Introduzione

Negli ultimi anni, la *sovranità digitale*, ovvero il mantenimento del controllo sui propri dati digitali, che essi siano di individui, imprese, o enti pubblici, sta diventando sempre più importante, soprattutto all'interno dell'Unione Europea, che per gli individui la ha già resa un pilastro fondamentale del software in Europa attraverso il *regolamento generale sulla protezione dei dati* (GDPR) [1], e, come uno dei suoi obiettivi per il futuro prossimo, sta proponendo di rafforzarla al fine di migliorare la resilienza dei servizi che le organizzazioni private e pubbliche dell'Unione offrono [2].

Avere il controllo sui propri dati però comporta delle sfide quando essi devono essere trasmessi altrove per essere elaborati, la prima delle quali é assicurare la compatibilità tra dati e metodi di elaborazione usati, richiedendo di stabilire a priori accordi sui processi utilizzati allo scopo [3], sia a livello legale, con legislazione apposita, sia a livello organizzazionale, stabilendo canali di comunicazione tra le parti coinvolte, sia a livello tecnico, definendo e implementando standard che rendano direttamente interoperabili tra loro più istanze di software anche eterogenei.

Nella mia tesi triennale «Progettazione e sviluppo di Sophon, applicativo cloud a supporto della ricerca» [4] e il successivo conference paper «Sophon: an Extensible Plat- form for Collaborative Research» [5], si era toccato il tema dell'interoperabilità fra i potenziali sviluppi futuri, menzionando che istituzioni utilizzanti *Sophon* avrebbero beneficiato dal poter comunicare con altre istanze di Sophon ospitate esternamente senza lasciare la propria.

Successivamente, Sophon si é evoluto in *NextPyter* [6], [7], ma l'interesse nell'arrivare all'interoperabilità é rimasto, con in mente il modello della *federazione di software*: fare in modo che diverse istanze di NextPyter potessero comunicare direttamente l'una con l'altra senza bisogno di un servizio centrale che coordini le loro interazioni.

Dei software utilizzanti il modello federato, uno dei più significativi é attualmente *Mastodon* [8], un software di microblogging che diffonde automaticamente i post di un utente su un'istanza di esso tra le istanze di quelli che lo seguono [9]: esiste, ad esempio, un istanza Mastodon ufficiale dell'Unione Europea [10], software sperimentali ad ausilio della ricerca che si integrano con esso, come Bonfire [11] ed Encyclia.pub [12], e progetti a supporto dello sviluppo software che stanno provando a implementare lo stesso modello per facilitare la collaborazione [13].

Si crede che parte del successo di Mastodon sia dovuto al suo utilizzo di standard di Internet già riconosciuti, tratti principalmente dagli *RFC* dell'IETF [14], [15] e dai *Web Standards* del W3C [16], [17], ottenendo grazie ad essi interoperabilità sia con i suoi predecessori e sia con i suoi potenziali successori, a costo di una significativa complessità progettuale.

Ritenendo questa complessità l'ostacolo più significativo allo sviluppo di ulteriori tecnologie nel campo dell'interoperabilità e federazione, si é scelto di lavorare per ridurla realizzando astrazioni specifiche allo scopo, con particolare attenzione a quelle richieste all'utilizzo di queste ultime in NextPyter, ma comunque proponendocisi di realizzarle

in modo che fossero riutilizzabili qualora i requisiti fossero cambiati, sufficientemente efficienti da essere utilizzabili anche da progetti su piccolissima scala, e con ottima integrazione con gli strumenti all'ausilio dello sviluppo software comunemente utilizzati dagli sviluppatori, come l'analisi statica del codice attraverso tipizzazione forte, e l'autocompletamento del codice che da essa deriva.

3. Concetti preliminari

3.1. Rust

Rust é un linguaggio di programmazione *general purpose* che pone particolare enfasi sull'efficienza e l'affidabilità dei programmi e sul migliorare la produttività degli sviluppatori che lo usano [18].

Per porre questa enfasi, si introducono il concetto di *proprietà dei valori* [19] e alcune regole associate ad esso, dette *regole di proprietà* (*ownership rules*), grazie alle quali il compilatore può minimizzare il numero di allocazioni dell'heap effettuate [20], impedire il verificarsi di situazioni di corsa (race condition) [21], ed effettuare ottimizzazioni che altrimenti non sarebbe possibile effettuare [22].

Il linguaggio sta trovando particolare successo nell'ambito dello sviluppo di servizi ed applicazioni di rete [23], sia a livello di client, in cui é utilizzato da alcune componenti del browser web *Firefox* [24] e nell'interezza del browser web *Servo* [25], sia a livello di server, in cui viene utilizzato da *Discord* [26], *1Password* [27], *Sentry* [28] e *Amazon Web Services* [29].

3.1.1. Proprietà dei valori

In Rust, ad ogni valore é associato **uno e un solo** *proprietario*, tipicamente una variabile, che ne determina la validità: fintanto che il proprietario é visibile nel blocco di codice in esecuzione, si é certi che il valore sia allocato, e viceversa, non appena il proprietario smette di essere visibile, si é certi che il programma deallocherà il valore [20].

Avere la proprietà di un valore significa quindi:
- ✓ poterlo leggere
- ✓ poterlo modificare
- ✓ poterne trasferire la proprietà
- ✓ potervi creare un riferimento mutabile [Sezione 3.1.2.]
- ✓ potervi creare un qualsiasi numero di riferimenti immutabili [Sezione 3.1.3.]

```rust
fn main() {
    let variabile = "Hello world!".to_string();
}
```

Codice A: La variabile `variabile` ha proprietà della stringa creata, e la stringa verrà deallocata al termine della funzione `main`.

3.1.2. Riferimenti mutabili

Prefissando l'operatore `&mut` al nome di una variabile, é possibile creare un *riferimento mutabile* al suo valore, in modo da permettere ad un altro blocco di codice di modificarne il valore, ma **senza trasferirgliene la proprietà**, evitando sia che il valore venga deallocato al termine del blocco che lo contiene, sia di dover trasferire esplicitamente la proprietà avanti e indietro tra i blocchi [21].

É però richiesto che, per ciascun valore, in ogni momento **non esista più di un riferimento mutabile ad esso**, perché altrimenti il valore potrebbe venire modificato é in uso e dare origine a una situazione di corsa [20].

Avere un riferimento mutabile a un valore significa quindi:
- ✓ poterlo leggere
- ✓ poterlo modificare
- **✗ non poterne trasferire la proprietà** [Sezione 3.1.1.]
- **✗ non potervi creare un riferimento mutabile**
- **✗ non potervi creare un riferimento immutabile** [Sezione 3.1.3.]

```
let mut variabile = "Hello world!".to_string();
let mutabile = &mut variabile;
```

Codice B: La variabile `mutabile` é un riferimento mutabile alla stringa creata.

3.1.3. Riferimenti immutabili

Prefissando l'operatore & al nome di una variabile, é invece possibile creare un *riferimento immutabile* al suo valore, in modo da permettere ad un altro blocco di codice di leggerne il valore, ma **senza permettergliene la modifica**, garantendo così che il valore non venga cambiato mentre sta venendo letto, e quindi rimuovendo il vincolo di avere un solo riferimento alla volta e permettendo letture concorrenti [21].

É però richiesto che mentre esistono riferimenti immutabili ad un valore, **non esistano un riferimento mutabile allo stesso**, o che venga modificato dal proprietario, perché altrimenti si tornerebbe nel caso di condizione di corsa [21].

Potendo avere più riferimenti immutabili allo stesso valore, Rust permette di duplicarli [30].

Avere un riferimento mutabile a un valore significa quindi:
- ✓ poterlo leggere
- **✗ non poterlo modificare**
- **✗ non poterne trasferire la proprietà** [Sezione 3.1.1.]
- **✗ non potervi creare un riferimento mutabile** [Sezione 3.1.2.]
- ✓ potervi creare un qualsiasi numero di riferimenti immutabili

```
let variabile = "Hello world!".to_string();
let immutabile = &variabile;
```

Codice C: La variabile `immutabile` é un riferimento immutabile alla stringa creata.

3.1.4. Lifetime

In un programma Rust corretto, i riferimenti, sia mutabili, sia immutabili, devono essere validi per tutta la durata della loro esistenza, ovvero il blocco di codice in cui si trovano: non possono esistere dopo che il valore a cui essi si riferiscono é stato deallocato [21].

La sezione di codice per cui un riferimento é valido é detta *lifetime*, e talvolta può essere annotata con un nome a cui é prefissato il glifo `'`, come ad esempio `'esempio` [31].

Per indicare il lifetime di un riferimento, se ne aggiunge il nome dopo `&`; un riferimento immutabile valido per `'lt` é quindi indicato con `&'lt valore`, mentre un riferimento mutabile valido per lo stesso é indicato con `&'lt mut valore`; in entrambi i casi, si dice che `valore` *vive per* o *é vivo per* `'lt` [31].

3.1.5. Crate

I programmi Rust sono raggruppati in blocchi di codice detti *crate*, concettualmente simili al concetto di *package* di altri linguaggi di programmazione, ma con la differenza chiave che **il compilatore Rust deve obbligatoriamente compilare una crate nel suo intero**, quando spesso altri linguaggi di programmazione sono in grado di compilare singoli file di un progetto alla volta [32].

Lavorare su crate troppo grandi può rappresentare un problema durante lo sviluppo, in quanto ogni singola modifica, benché piccola, richiede la ricompilazione dell'intera crate.

3.1.6. Trait

All'interno di una crate [Sezione 3.1.5.] é possibile definire insiemi di funzioni detti *trait* che un tipo può scegliere di *implementare* (`impl`) in modo arbitrario per acquisire quel *trait*, in modo simile al concetto di *interfaccia* della programmazione a oggetti [33].

Una volta definiti, é poi possibile utilizzare uno o più trait per introdurre restrizioni sui tipi che possono essere usati al posto di un parametro generico [33].

Ad esempio, il metodo `.dedup()`, che rimuove i duplicati da un vettore (`Vec`), può essere chiamato solo se il tipo usato per gli elementi del vettore implementa `PartialEq`, ovvero la possibilità di due elementi dello stesso tipo di essere uguagliati [34].

```
struct CiaoMondo;

trait TraducibileInInglese {
    fn traduci(self) -> String;
}

impl TraducibileInInglese for CiaoMondo {
    fn traduci(self) -> String {
        "Hello world".to_string()
    }
}
```

Codice D: Esempio di definizione di un trait e implementazione su una struct.

3.1.7. Associated type

Un trait può avere uno o più *associated types*, tipi «astratti» che vengono definiti nel momento della definizione del trait, e dopo specificati nel momento della sua implementazione [35].

A differenza di un parametro generico, un tipo associato é invariabile per quella specifica implementazione del trait, quindi se un trait é implementato per un dato tipo, per quel tipo il tipo associato sarà sempre lo stesso.

Un esempio di tipo associato nel trait `FromStr`, che permette di costruire un'istanza del tipo che lo implementa partendo da una stringa, é `FromStr::Err`, ovvero il tipo dell'errore restituito qualora la costruzione dell'istanza fallisse; per i numeri a virgola fissa (come `i32`), esso é specificato come `ParseIntError`, mentre per i numeri a virgola mobile (come `f32`), esso é specificato come `ParseFloatError`, e non é mai data l'opportunità all'utilizzatore di quelle implementazioni di selezionarne uno diverso [36].

```
struct CiaoMondo;
struct ErroreTraduzione;

trait TraducibileInInglese {
    type Errore;

    fn prova_a_tradurre(self) -> Result<String, Self::Errore>;
}

impl TraducibileInInglese for CiaoMondo {
    type Errore = ErroreTraduzione;

    fn prova_a_tradurre(self) -> Result<String, Self::Errore> {
        ErroreTraduzione
    }
}
```

Codice E: Estensione dell'esempio di [Codice D] a cui si é aggiunto l'associated type `Errore`.

3.1.8. Orphan rules

Per evitare che uno stesso trait possa essere implementato per lo stesso tipo in modi diversi, il compilatore Rust impone restrizioni sulle implementazioni.

Le restrizioni per i casi specifici sono complesse [37], ma nella grande maggioranza dei casi, un tipo può implementare un trait solo se:
- o il tipo, o il trait sono definiti nella crate in cui sta venendo implementato;
- non esistono altre implementazioni dello stesso trait nella crate.

3.1.9. Item

Un *item*, in Rust, é qualsiasi componente di codice dichiarato nello scope immediatamente interno ad un modulo [38].

Sono item ad esempio [38]:
- costanti
- blocchi di definizione di funzione `fn`

- blocchi di definizione di strutture `struct`
- blocchi di definizione di tratti `trait` [Sezione 3.1.6.]
- blocchi di implementazione di tratti `impl` [Sezione 3.1.6.]

3.1.10. Cargo

Cargo é il package manager di Rust, ovvero il software che si occupa di scaricare e mantenere aggiornate le crate [Sezione 3.1.5.] indicate come dipendenze di un progetto, e di orchestrare formattazione, linting, compilazione, linking, testing e pubblicazione di tutte le crate in esso [39].

Per determinare cosa fare, Cargo legge il file `Cargo.toml` presente nella working directory: quel file, detto il *manifest*, contiene informazioni sulle crate in essa contenute [40].

Un progetto Cargo può essere composto da:
- una singola crate (é il caso della maggior parte delle crate);
- più crate definite in uno stesso `Cargo.toml`, con la limitazione che una sola di esse può essere una libreria (é il caso delle librerie che includono uno o più eseguibili);
- un *workspace* contenente più sottocartelle, ciascuna contenente un proprio `Cargo.toml` che definisce una o più crate secondo le due regole sopra (é il caso di quando più librerie collegate vengono sviluppate parallelamente dagli stessi sviluppatori).

3.1.11. Feature

Una crate può avere delle *feature*, funzionalità opzionali che é possibile decidere se includere oppure no nel software al momento della compilazione [41].

All'interno del file `Cargo.toml`, é possibile definire le feature che mette a disposizione la crate attuale, e quali feature delle dipendenze dovranno essere compilate perché la dipendenza sia soddisfatta [41].

3.1.12. Macro

In certi casi, il codice Rust può risultare essere molto verboso; ad esempio, anche solo implementare trait di uso comune come `Clone` o `PartialEq` risulterebbe in almeno 5 righe di codice aggiuntive per ciascuna implementazione.

Per evitare questo, Rust ha la possibilità di definire e usare *macro*: elementi di codice che si *espandono* ad elementi di codice più grandi al momento della compilazione [42].

Esistono quattro tipi di macro [42]:
- funzioni macro dichiarative (descritte in [Sezione 3.1.13.])
- funzioni macro procedurali (non usate in questa tesi)
- derivazioni macro procedurali (menzionate in [Sezione 3.1.15.])
- attributi macro procedurali (descritti in [Sezione 3.1.14.])

Le macro procedurali possono essere definite solo in crate dedicate del tipo `proc-macro`, ma possono poi essere usate da qualsiasi altra crate [43].

3.1.13. Funzione macro dichiarativa

Una *funzione macro dichiarativa* [42] o *macro-by-example* [44] o *macro_rules!* [45] é composta da un nome, argomenti in forma di uno o più pattern di token, e per ogni possibile pattern di argomenti, il blocco di codice che sarà generato in corrispondenza di essi [42].

```
macro_rules! mia_macro {
  ($tipo:ty, $nome:ident) => {
    let $nome: $tipo = 1;
  }
}
```

Codice F: Definizione di una funzione macro dichiarativa di esempio con `macro_rules!`.

Una volta definita, una funzione macro dichiarativa può essere inserita ed espansa nel codice in qualsiasi posizione scrivendo il suo nome, seguito da un punto esclamativo, seguito dagli argomenti circondati da parentesi tonde, quadre, o graffe [44].

Un esempio é la macro `vec`, usabile per creare un vettore `Vec` da una serie di elementi [46].

```
vec!["primo", "secondo", "terzo"];
```

Codice G: Una chiamata alla funzione macro dichiarativa `Vec`.

Non eseguendo codice, ma solo sostituendolo con blocchi di codice corrispondenti agli argomenti, hanno capacità limitate: ad esempio, non possono consultare fonti esterne ai loro argomenti o alla loro definizione per determinare cosa sarà generato.

3.1.14. Attributo macro procedurale

Un *attributo macro procedurale* é definito come una funzione marcata con l'attributo `#[proc_macro_attribute]` in una crate di tipo `proc-macro` che riceve due `TokenStream` come argomenti e ne emette un altro in output [43].

```
#[proc_macro_attribute]
fn mia_macro(attr: TokenStream, item: TokenStream) -> TokenStream {
  // ...
}
```

Codice H: Definizione di un attributo macro procedurale di esempio con `#[proc_macro_attribute]`.

Una volta definita, la macro può essere applicata a qualsiasi item marcandolo con un attributo con il nome della macro; al momento della compilazione, la funzione precedentemente definita sarà eseguita, e l'item sarà sostituito con il `TokenStream` emesso in output da essa [43].

Il primo `TokenStream` ricevuto in input dalla funzione corrisponde al contenuto dell'attributo successivo al nome, mentre il secondo corrisponde al contenuto dell'item a cui é stato applicato l'attributo [43].

```
#[crate_con_le_macro::mia_macro(questo_viene_passato_ad_attr)]
struct QuestoVienePassatoAdItem {}
```

Codice I: Un utilizzo dell'attributo macro procedurale definito in [Codice H].

3.1.15. Crate `serde`

`serde` é una crate Rust che permette la *ser*-ializzazione e la *de*-serializzazione efficiente di strutture dati Rust, generalizzata a vari formati di interscambio dati [47].

Tra i formati di interscambio dati che supporta, troviamo [47]:
- URL Query string [Sezione 3.2.6.]
- XML [Sezione 3.4.1.]
- JSON [Sezione 3.5.1.]

Per implementare automaticamente serializzazione e deserializzazione con `serde`, é sufficiente aggiungere la crate e la sua funzionalità `derive` alle dipendenze, e poi aggiungere le derivazioni macro procedurali `Serialize` e `Deserialize` alle strutture dati che si desidera rispettivamente serializzare e deserializzare [48].

```
use serde::Serialize;
use serde::Deserialize;

#[derive(Serialize, Deserialize)]
struct Point {
    x: i32,
    y: i32,
}
```

Codice J: Definizione di una struttura dati con supporto a serializzazione e deserializzazione attraverso `serde`.

É una delle crate più utilizzate dell'ecosistema Rust [49].

3.1.16. Crate `thiserror`

`thiserror` é una crate Rust che permette di implementare il tratto `std::error::Error`, rappresentante la capacità di un tipo di essere rappresentato come errore, attraverso la derivazione macro procedurale `thiserror::Error` che essa fornisce [50].

Tipicamente viene applicata agli `enum`, facendo in modo che ad ogni variante dell'`enum` corrisponda un messaggio di errore assegnatole con l'attributo `#[error(...)]` [50].

```rust
use thiserror::Error;

#[derive(Debug, Error)]
enum MyError {
    #[error("this thing caused a problem")]
    ThisProblem,

    #[error("that thing caused a problem: {:?}", .0)]
    ThatProblem(&Thing),
}
```

Codice K: Definizione di un enum rappresentante un errore di esempio che potrebbe essere causato da «questa cosa» o da «quella cosa», con un riferimento alla «cosa» che l'ha causato nel secondo caso. Il tratto `Debug` della «cosa» nel secondo caso viene usato per rappresentarla nel messaggio di errore.

3.1.17. Crate `log`

`log` é una crate Rust che permette di implementare logging a più livelli astraendo sul formato in cui i log vengono emessi [51].

Utilizzando `log`, é possibile generare il codice per l'emissione di un log usando la stessa sintassi usata per `println!`, ma usando invece una delle funzioni macro [Sezione 3.1.12.] incluse nella crate, che in ordine di importanza sono: [51]

- `error!(...)`
- `warn!(...)`
- `info!(...)`
- `debug!(...)`
- `trace!(...)`

```rust
log::warn!("Qualcosa sta andando storto: {questo:?}")
```

Codice L: Esempio di chiamata a `warn!` utilizzata per emettere un avvertimento relativo alla variabile `questo`.

Fa uso delle feature [Sezione 3.1.11.] per determinare quali istruzioni di logging devono essere incluse nell'eseguibile finale: ad esempio, abilitare la feature `release_max_level_warn` eliminerà tutte le chiamate a `info!`, `debug!` e `trace!` dalle build ottimizzate «Release» [51].

3.1.18. Crate `pretty_env_logger`

`pretty_env_logger` é un'implementazione di `log` [Sezione 3.1.17.] che legge il valore della variabile d'ambiente `RUST_LOG` per determinare la configurazione desiderata dei log, poi li emette in standard output, formattandoli con tabulazione e colori [52].

Si inizializza inserendo una chiamata a `pretty_env_logger::init` all'inizio del codice [52].

```
export RUST_LOG="mycrate=debug"
```

Codice M: Esempio di configurazione che emette su standard output tutti i messaggi di importanza `debug` o superiore provenienti dalla crate `mycrate`.

3.2. Servizi web

Un *servizio web* é un sistema software che permette interoperabilmente interazioni machine-to-machine attraverso una rete [53].

3.2.1. Agenti

Le due parti software che si scambiano messaggi in un servizio web sono dette *agenti*; quella che effettua la richiesta é detta l'*agente richiedente*, mentre quella che la riceve é detta l'*agente fornitore* [53].

3.2.2. Risorse

Un servizio web é composto da una o più *risorse*, identificabili univocamente da un URL [53], ciascuna che permette all'agente richiedente di effettuare una determinata operazione.

```
https://codeberg.org/forgejo/forgejo
```

Codice N: Una risorsa di un servizio web. Offre accesso al repository di codice di Forgejo.

3.2.3. Content negotiation

Uno dei modi in cui é possibile ottenere questa interoperabilità é attraverso la *content negotiation*: specificando l'header `Accept` in una richiesta HTTP, é possibile informare un servizio web di quali formati sono supportati dall'agente richiedente, in modo che l'agente fornitore possa selezionare uno di quelli per inviare la risposta [54].

```
GET ...
...
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
...
```

Codice O: Header `Accept` inviato da Zen Browser per richiedere pagine web. Manifesta una preferenza per HTML, seguita da XHTML, poi XML, e infine qualsiasi altro formato supportato.

Servizi web che per la stessa risorsa [Sezione 3.2.2.] restituiscono documenti diversi in base al valore dell'header `Accept` devono specificare l'header `Vary` nella risposta con l'esatto valore `Accept`, notificando così eventuali cache che esso deve essere preso in considerazione [54].

3.2.4. Crate `axum`

La crate Rust `axum` é un framework per la creazione di servizi web (chiamati «applicazioni web» nella documentazione) in modo modulare ed ergonomico [55].

3.2.5. Handler `axum`

`axum` [Sezione 3.2.4.] fa uso dei trait e tipi generici di Rust per rendere possibile la definizione di risorse attraverso funzioni, dette *handler*, che vengono eseguite per effettuare l'operazione richiesta ed emettere la relativa risposta [55].

3.2.6. Extractor `axum`

Un handler `axum` [Sezione 3.2.5.] può avere zero o più argomenti, detti *extractor*, che processano i dati della richiesta HTTP ricevuta e li trasformano in strutture dati Rust appropriate [56].

I tipi degli extractor devono implementare uno a scelta dei tratti `FromRequestParts` o `FromRequest`: in un handler possono esserci illimitati extractor con tipi che implementano il primo, in quanto essi ricevono solo una reference immutabile [Sezione 3.1.3.] al corpo della richiesta HTTP, ma **solo uno** che implementa il secondo, in quanto esso riceve la proprietà [Sezione 3.1.1.] del corpo [57].

Alcuni tipi di extractor comuni sono [56]:
- `Path<T>`, che ricava un valore di tipo `T` dal path della richiesta HTTP;
- `Query<T>`, che ricava una mappa chiave-valore di tipo `T` dalla query string della richiesta HTTP;
- `HeaderMap`, che permette accesso agli header della richiesta HTTP;
- `String`, che restituisce il corpo della richiesta HTTP come stringa Rust;
- `Json<T>`, che utilizza `serde_json` [Sezione 3.5.1.] per effettuare il parsing del corpo della richiesta HTTP e ricavarne il tipo `T`;
- `Request`, che restituisce l'oggetto corrispondente alla richiesta HTTP, lasciando all'handler completa libertà;
- `Extension<T>`, che non interagisce con la richiesta HTTP ma permette di accedere ad una struttura dati di tipo `T` creata nel momento della definizione del server `axum`, tipicamente usato per condividere funzionalità tra handler diversi, come connessioni a database [58].

3.2.7. Response `axum`

Un handler `axum` [Sezione 3.2.5.] può restituire qualsiasi valore il cui tipo implementi il trait `IntoResponse`, il cui metodo `into_response` si occupa di riconvertirlo nella risposta HTTP da servire [59].

Tipi di *response* comuni sono [60]:
- `StatusCode`, che restituisce una risposta vuota con l'HTTP status code specificato;
- `String`, utilizza la stringa specificata come corpo della risposta HTTP;
- `(StatusCode, String)`, che combina `StatusCode` con `String`;

- `Json<T>`, che utilizza `serde_json` [Sezione 3.5.1.] per serializzare il tipo `T` in un valore JSON appropriato che sarà usato come corpo della risposta;
- `(StatusCode, Json<T>)`, che combina `StatusCode` con `Json<T>`;
- `Result<Ok, Err>`, che si comporta come la response `Ok` se tutto ha funzionato correttamente, o come la response `Err` se si é verificato un errore.

3.2.8. Crate `minijinja`

`minijinja` é una crate Rust che fornisce capacità di template engine molto simili a quelle che la libreria `jinja2`, usata in Flask [61], fornisce in Python, basandosi su `serde` [Sezione 3.1.15.] per processare i valori dati al template [62].

```
{% for user in users %}
  <li>{{ user.name }}</li>
{% endfor %}
```

Codice P: Esempio di template che `minijinja` riporta nella sua documentazione.

É facile da integrare con `axum`, aggiungendogli così la capacità avanzate di generazione di pagine HTML al momento delle richieste.

Prima di essere usata, richiede l'inizializzazione di un `minijinja::Environment`, all'interno del quale andranno inseriti tutti i template utilizzabili [62].

```rust
use minijinja::{Environment, context};

let mut env = Environment::new();
env.add_template("hello", "Hello {{ name }}!").unwrap();
```

Codice Q: Esempio di inizializzazione di un `minijinja::Environment` che `minijinja` riporta nella sua documentazione. Il primo parametro di `add_template`, `"hello"`, é il nome con cui quel template sarà referenziabile all'interno di altri template, mentre il secondo parametro é il contenuto non valutato del template stesso.

3.3. cURL

cURL, o *Client for URLs*, é un software che, dato un URL di uno dei protocolli supportati, é in grado di recuperare la risorsa a quell'URL [63], [64].

Ha una diffusione vastissima: é distribuito per tutti i maggiori sistemi operativi, ed é facile trovarlo incluso nei sistemi informatici più disparati, come automobili, televisioni, o stampanti [65].

```
curl 'https://example.org'
```

Codice R: cURL utilizzato per inviare una richiesta HTTPS GET a `example.org`.

3.4. XML

XML, abbreviazione di *Extensible Markup Language*, é un formato per la pubblicazione digitale che ha successivamente trovato successo nell'essere anche un formato di interscambio dati [66].

Fa uso di una sintassi estremamente simile, ma non completamente uguale, a quella usata da *HTML*, il linguaggio usato per strutturare pagine web.

Essendo stato raccomandato dal *W3C* (*World Wide Web Consortium*) [67], e usato nel protocollo *SOAP* [68], é diventato uno dei principali formati di interscambio dati del web, ma ha recentemente perso popolarità in favore di JSON [Sezione 3.5.], più semplice da scrivere e processare.

3.4.1. Crate `quick-xml`

La crate Rust `quick-xml` implementa (fra le sue varie funzionalità) la serializzazione e deserializzazione di `serde` [Sezione 3.1.15.] nel formato XML [69].

Per via di alcune differenze tra lo standard XML e il modello dati di `serde`, é possibile che per utilizzarla per deserializzare documenti XML esistenti sia necessario configurare le strutture dati in cui si deriva `Serialize` e `Deserialize` in modo insolito [70].

3.5. JSON

JSON, abbreviazione di *JavaScript Object Notation* (*Notazione Oggetti JavaScript*), é un formato di interscambio dati che si propone di essere al tempo stesso semplice da leggere e scrivere per le persone e facile da processare e generare per i computer [71].

Fa uso della stessa sintassi che JavaScript usa per definire nuovi oggetti, da cui il nome JavaScript Object Notation.

La sua ubiquità [71] e facile integrazione in JavaScript [72] lo hanno reso il formato di interscambio dati preferito da molti servizi web [Sezione 3.2.].

3.5.1. Crate `serde_json`

La crate Rust `serde_json` implementa la serializzazione e deserializzazione di `serde` [Sezione 3.1.15.] nel formato JSON [73].

Non é richiesto nulla di particolare per utilizzarla; per processare JSON, é sufficiente utilizzare i metodi di serializzazione e deserializzazione che fornisce [73].

3.6. PostgreSQL

PostgreSQL é un sistema open source per database oggetto-relazionali che si basa su ed estende il linguaggio SQL [74].

3.6.1. Crate `diesel`

La crate `diesel` é un modello oggetto-relazionale per Rust che sfrutta il sistema di tipizzazione di Rust per offrire la possibilità di interagire con un database in maniera sicura e componibile [75].

Fa uso di un tool esterno al codice compilato, `diesel_cli`, per interrogare il database in esecuzione e da esso generare attraverso funzioni macro dichiarative [Sezione 3.1.13.] varie astrazioni Rust per permettervi l'interazione [76].

Include un sistema per migrazioni integrabile direttamente negli eseguibili prodotti che permette l'aggiornamento graduale del database alla forma che lo schema prende durante lo sviluppo senza perdere i dati già archiviati in esso [76].

3.6.2. Crate `diesel_async`

La crate `diesel_async` estende `diesel` permettendo di interagire con il database utilizzando funzioni asincrone [77].

A parte il requisito di usare funzioni `async`, chiamate `await`, e tratti da `diesel_async` anziché `diesel` per l'interazione, la sua interfaccia é fondamentalmente uguale a quella di `diesel` [77].

3.7. Docker

Docker é una piattaforma per facilitare lo sviluppo, distribuzione, ed esecuzione di applicazioni attraverso l'unificazione degli ambienti software in cui il codice viene eseguito [78].

Astrae vari aspetti degli ambienti software, come sistema operativo o volumi di archiviazione, ma ai fini di questa tesi si evidenziano tre concetti:
- immagini Docker [Sezione 3.7.1.]
- container Docker [Sezione 3.7.2.]
- progetti Docker Compose [Sezione 3.7.3.]

3.7.1. Immagini Docker

Una *immagine Docker* é un insieme immutabile di elementi che descrive come é composto, come funziona, e come fare funzionare un ambiente software [78].

Le immagini sono impacchettate in modo tale da poter essere facilmente redistribuite e composte tra loro, al punto da rendere la composizione un concetto quasi onnipresente nell'ecosistema Docker [79].

Ad esempio:
1. una applicazione Rust avrà una sua immagine;
2. che si baserà sull'immagine ufficiale Rust per la compilazione [80];
3. che a sua volta si baserà sull'immagine ufficiale Debian per le funzionalità del sistema operativo [81].

Un'immagine é creata a partire da un `Dockerfile`, che contiene la serie di istruzioni imperative necessarie per arrivare al prodotto finale [82].

```
FROM rust
WORKDIR /usr/src/miaapp
COPY . .
RUN cargo build --release
ENTRYPOINT ["miaapp"]
```

Codice S: Breve Dockerfile non ottimizzato che compila un'applicazione Rust e la rende utilizzabile nell'immagine risultante.

3.7.2. Container Docker

Un *container Docker* é un'istanza di una immagine Docker, tipicamente quella di un'applicazione finale, in esecuzione in isolamento dal resto del sistema operativo che la sta ospitando [83].

É possibile concedere a due o più container Docker di comunicare tra loro; l'esempio tipico é quello di un'applicazione che necessita di un database, il cui container viene connesso ad un altro container che sta eseguendo PostgreSQL [Sezione 3.6.].

3.7.3. Progetti Docker Compose

Un *progetto Docker Compose* é una directory contenente un file `compose.yml` in cui sono specificati dichiarativamente una serie di container Docker e i modi in cui essi devono essere configurati e connessi, permettendo all'utente di avviare, terminare, o monitorare un blocco di container alla volta [84].

```yaml
services:
  database:
    container_name: &database "database"
    image: "postgres:17"
    restart: unless-stopped
    volumes:
      - # Database data
        type: bind
        source: "./data/database"
        target: "/var/lib/postgresql/data"
    environment:
      # Allow connections from this compose stack
      POSTGRES_HOST_AUTH_METHOD: "trust"
      POSTGRES_DB: &database_name "miaapp"
      POSTGRES_USER: &database_user "miaapp"

  miaapp:
    build: "./src/miaapp"
    restart: unless-stopped
    environment:
      POSTGRES_HOST: *database
      POSTGRES_DB: *database_name
      POSTGRES_USER: *database_user
      MIAAPP_PORT: &miaapp_port 80
    ports:
      - name: "http"
        target: *miaapp_port
        host_ip: "127.0.0.1"
        published: 30000
        protocol: tcp
        app_protocol: http
```

Codice T: Progetto Docker Compose di esempio per creare l'immagine di un'applicazione ed eseguirla connettendola a un database PostgreSQL istanziato appositamente.

3.7.4. Crate `micronfig`

La crate Rust `micronfig`, da me realizzata precedentemente e separatamente da questa tesi, permette di definire parametri di configurazione per un programma Rust attraverso una singola chiamata ad una funzione macro dichiarativa che essa fornisce, `config!` [85].

```
micronfig::config! {
    DATABASE_URI,
    APPLICATION_NAME: String,
    MAX_CONCURRENT_USERS: String > u64,
    SHOWN_ALERT?,
}
```

Codice U: Esempio di utilizzo di `config!` fornito nella documentazione di `micronfig`. Il valore di `MAX_CONCURRENT_USERS` viene convertito a `u64` al momento del primo accesso, mentre il valore di `SHOWN_ALERT` é una `String` ed é opzionale.

É pensata per permettere la facile configurazione di binari Rust in diversi ambienti, come in esecuzione sul computer locale, all'interno di una propria immagine Docker [Sezione 3.7.1.], un progetto Docker Compose [Sezione 3.7.3.], o anche contesti più complessi come Docker Swarm o Kubernetes, supportando una gerarchia di fonti da cui recuperare i valori configurati [85].

3.8. Resource descriptor

Un *resource descriptor* é un documento di metadati che descrivono quali risorse [Sezione 3.2.2.] sono messe a disposizione da un servizio web [Sezione 3.2.], o, più genericamente, da un host [86].

Fanno uso di resource descriptor il protocollo di messaggistica XMPP [87] e il protocollo WebFinger [88], che a sua volta é utilizzato da Mastodon [15] e OpenID Connect [89].

Un resource descriptor può avere uno di due formati, selezionabili attraverso content negotiation [Sezione 3.2.3.] o a percorsi diversi:

- *XML Resource Descriptor*, o *XRD*, con il media type `application/xrd+xml` o percorso `/.well-known/host-meta` [86]
- *JSON Resource Descriptor*, o *JRD*, con il media type `application/jrd+json` o percorso `/.well-known/host-meta.json` [88]

```
{
  "links": [
    {
      "rel": "lrdd",
      "template": "https://mastodon.social/.well-known/webfinger?
resource={uri}"
    }
  ]
}
```

Codice V: Il file host-meta di `mastodon.social`, che descrive l'URL a cui devono essere effettuate richieste WebFinger.

3.9. NodeInfo

Un documento *NodeInfo* é un documento JSON [Sezione 3.5.] contenente specifici metadati riguardanti gli utenti di un servizio web, o più nello specifico, un sito web di social networking distribuito [90].

Fra i metadati che prevede, si possono trovare:

- nome del sito [91]
- nome, versione, e repository di codice del software che sta eseguendo [91]
- conteggio degli utenti registrati, attivi negli ultimi 6 mesi, 1 mese, e 1 settimana [91]
- conteggio dei post e commenti presenti sul sito [91]

Lo scopo principale dei documenti NodeInfo é quello di pubblicare statistiche in un formato omogneo riguardanti servizi web eterogenei ma intercompatibili, in modo che possano poi essere aggregate da servizi appositi per ottenere informazioni sullo stato della rete intercompatibile [8], [92], [93].

```json
{
  "version": "2.0",
  "software": {
    "name": "mastodon",
    "version": "4.4.0-nightly.2025-06-17"
  },
  "protocols": [
    "activitypub"
  ],
  "services": {
    "outbound": [],
    "inbound": []
  },
  "usage": {
    "users": {
      "total": 2772800,
      "activeMonth": 281126,
      "activeHalfyear": 751154
    },
    "localPosts": 135285543
  },
  "openRegistrations": true,
  "metadata": {
    "nodeName": "Mastodon",
    "nodeDescription": "The original server operated by the Mastodon gGmbH non-profit"
  }
}
```

Codice W: Il file NodeInfo di `mastodon.social`, recuperato il 2025-06-17.

3.10. JSON-LD

JSON-LD, abbreviazione di *JSON for Linked Data* (*JSON per Dati Interconnessi*), é un'estensione al formato JSON che propone una migliore organizzazione, standardizzazione, e scopribilità dei dati all'interno dei siti web, mantenendo allo stesso tempo la facile leggibilità che caratterizza il formato JSON e permettendo un facile aggiornamento da JSON a JSON-LD [94], [95].

Le funzionalità chiave che JSON-LD aggiunge a JSON sono [95]:
- identificazione universale degli oggetti JSON attraverso IRI [Sezione 3.10.1.]
- valori JSON che possono riferirsi a una risorsa remota attraverso IRI ([Sezione 3.10.1.], [Sezione 3.2.2.])
- rappresentazione di grafi diretti in un singolo documento [Sezione 3.10.2.]
- disambiguazione chiavi associate a documenti JSON diversi attraverso IRI e contesti [Sezione 3.10.3.]
- annotazione di stringhe con la loro lingua e direzione di scrittura ([Sezione 3.10.4.], [Sezione 3.10.5.])
- associazione di valori JSON a tipi astratti come date o durate [Sezione 3.10.6.]

Trova utilizzo in disparati ambiti, dalle email [96], alla demografia [97], al social networking [Sezione 3.11.].

3.10.1. IRI

Un *IRI*, *Internationalized Resource Identifier* (*Identificatore di Risorse Internazionalizzato*), é un'alternativa ai più comuni URI che permette l'utilizzo di tutti i caratteri Unicode [98] anziché solo i caratteri US-ASCII [99], rendendo così gli identificatori in lingue con alfabeti non-latini comunque facilmente leggibili e modificabili da persone.

```
https://en.wiktionary.org/wiki/Ῥόδος
```

Codice X: IRI della pagina di Wiktionary del nome proprio «Rodi» in greco antico.

```
https://en.wiktionary.org/wiki/%E1%BF%AC%CF%8C%CE%B4%CE%BF%CF%82
```

Codice Y: URI equivalente all'IRI in [Codice X].

3.10.2. Forme

JSON-LD permette di rappresentare lo stesso documento in *forme* diverse, ciascuna con vantaggi e svantaggi diversi [95].

La forma più comune é quella *compatta*, che cerca di massimizzare l'utilizzo di contesto [Sezione 3.10.3.], rendendo così il documento più breve e di più facile lettura umana [95].

```
{
  "@context": "http://schema.org/",
  "@type": "Person",
  "name": "Jane Doe",
  "jobTitle": "Professor",
  "telephone": "(425) 123-4567",
  "url": "http://janedoe.example.org"
}
```

Codice Z: Alcune informazioni riguardanti una persona fittizia, rappresentate in JSON-LD con forma compatta.

Sviluppatori di applicazioni facenti uso di JSON-LD si troveranno a lavorare spesso con la forma *espansa*, che rimuove tutto il contesto [Sezione 3.10.3.] scrivendo per esplicito chiavi, tipi, e valori [95].

```
[
  {
    "@type": [
      "http://schema.org/Person"
    ],
    "http://schema.org/jobTitle": [
      {
        "@value": "Professor"
      }
    ],
    "http://schema.org/name": [
      {
        "@value": "Jane Doe"
      }
    ],
    "http://schema.org/telephone": [
      {
        "@value": "(425) 123-4567"
      }
    ],
    "http://schema.org/url": [
      {
        "@id": "http://janedoe.example.org"
      }
    ]
  }
]
```

Codice AA: Le stesse informazioni in [Codice Z], rappresentate in forma espansa. Di particolare rilevanza é il fatto che tutte le proprietà, sebbene in forma contratta fossero scalari, siano state trasformate in vettori di cardinalità 1.

Un altra forma possibile é quella *appiattita*, che rappresenta tutti i dati del documento come un singolo grafo diretto; in certi casi, questo potrebbe semplificare l'elaborazione richiesta [95].

```
{
  "@context": "http://schema.org/",
  "@graph": [
    {
      "id": "_:b0",
      "type": "Person",
      "jobTitle": "Professor",
      "name": "Jane Doe",
      "telephone": "(425) 123-4567",
      "url": "http://janedoe.example.org"
    }
  ]
}
```

Codice AB: Le stesse informazioni in [Codice Z], rappresentate in forma appiattita.

3.10.3. Contesto

Esplicitare manualmente tutti gli IRI [Sezione 3.10.1.] é molto tedioso per un autore di un documento JSON-LD, e di difficile lettura per il suo destinatario [95].

Per ovviare a questo problema, JSON-LD permette di specificare il *contesto* del documento: un altro documento JSON-LD, anche remoto, che associa brevi *termini* a corrispondenti IRI [95].

```
{
  "@context": {
    "ste": "https://meta.steffo.eu/"
  },
  "ste": "ffo"
}
```

Codice AC: Documento JSON-LD di esempio con contesto che associa il termine `ste` all'iri `https://meta.steffo.eu/`.

Il contesto é specificato dal valore della chiave `@context` di ciascun oggetto nel documento, oppure, se si tratta di un documento recuperato via web, dal valore dell'header `Link` della risposta [95].

```
HTTP/1.1 200 OK
...
Content-Type: application/json
Link: <https://json-ld.org/contexts/person.jsonld>; rel="http://www.w3.
org/ns/json-ld#context"; type="application/ld+json"
...
```

Codice AD: Esempio di header `Link` impostato per applicare un contesto al documento JSON restituito.

3.10.4. Lingua delle stringhe

Come già menzionato in precedenza, JSON-LD permette l'associazione di una lingua a ciascun valore stringa presente nei documenti: questo avviene attraverso il valore della chiave `@language` [95].

Essa può essere specificata nel contesto [Sezione 3.10.3.] per impostare la lingua a tutte le stringhe presenti in esso o al valore di uno specifico termine, o anche direttamente all'interno del valore stesso, sostituendolo con un oggetto JSON con la chiave `@language` impostata al valore desiderato e il valore della stringa inserito alla chiave `@value` [95].

```
{
  "@context": {
    "@language": "it-it"
  },
  "https://meta.steffo.eu/testo": "Ciao mondo!"
}
```

Codice AE: Documento JSON-LD con la lingua globalmente impostata a Italiano.

```
{
  "@context": {
    "ste": "https://meta.steffo.eu/",
    "ste:testo": {
      "@language": "it-it"
    }
  },
  "ste:testo": "Ciao mondo!"
}
```

Codice AF: Documento JSON-LD in cui la lingua del valore di `https://meta.steffo.eu/testo` é impostata a Italiano attraverso il contesto.

```
{
  "https://meta.steffo.eu/testo": {
    "@value": "Ciao mondo!",
    "@language": "it-it"
  }
}
```

Codice AG: Documento JSON-LD privo di contesto in cui la lingua del valore di `https://meta.steffo.eu/testo` é impostata a Italiano direttamente dal valore.

Inoltre, é possibile specificare *language maps*, oggetti JSON utilizzabili come valori, aventi language tags ([100]) come chiavi e stringhe in quella lingua come valori, ottenendo così una proprietà con valori diversi in lingue diverse [95].

```
{
  "@context": {
    "ste": "https://meta.steffo.eu/",
    "ste:testo": {
      "@container": "@language"
    }
  },
  "ste:testo": {
    "it-it": "Ciao mondo!",
    "en-us": "Hello world!"
  }
}
```

Codice AH: Documento JSON-LD contenente una language map nella proprietà `https://meta.steffo.eu/testo`, rendendo disponibile il valore di essa sia in Italiano, sia in Inglese (Statunitense).

3.10.5. Direzione di scrittura

Lo stesso concetto di `@language` [Sezione 3.10.4.], con l'eccezione delle language maps, si applica per `@direction`, specificando così la direzione di lettura dei valori, `ltr` (da sinistra a destra) e `rtl` (da destra a sinistra) [95].

3.10.6. Tipi

Anche se JSON supporta un insieme limitato di tipi di valori, JSON-LD permette di rappresentare all'interno dei suoi documenti valori di qualsiasi *tipo*, come numeri interi, date, o durate [95].

Questo avviene tramite la chiave `@type`, specificata negli stessi modi in cui é specificabile `@direction` [Sezione 3.10.5.], il cui valore deve però essere un IRI [Sezione 3.10.1.], o un termine riconducibile ad esso tramite il contesto [Sezione 3.10.3.] [95].

Tra i tipi comunemente usati, si evidenziano quelli dell'XML Schema definito dal W3C, che include tanti tipi di uso comune non disponibili in JSON come `date`, `time`, `dateTime`, `duration`, `decimal`, `nonNegativeInteger`, e `anyURI` [101].

```
{
  "@context": {
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "ste": "https://meta.steffo.eu/#",
    "ste:duration": {
      "@type": "xsd:duration"
    }
  },
  "@type": "ste:Sleep",
  "ste:duration": "P9H"
}
```

Codice AI: Documento JSON-LD di esempio che rappresenta un sonno dalla durata di 9 ore, rappresentata attraverso il tipo XML Schema `duration`.

3.10.7. Crate `json_ld`

La crate Rust `json_ld` permette di elaborare documenti JSON-LD all'interno di programmi Rust, con supporto a operazioni di deserializzazione, lettura, scrittura, serializzazione, validazione, compattazione, espansione, appiattificazione, e in generale di cambio forma [102].

É molto potente, ma produce codice molto verboso; anche solo impostare una chiave a una stringa può richiedere di passare attraverso tanti diversi item [103], [104], [105], [106], [107]!

```
Indexed::new(
  Object::Value(
    Value::Literal(
      Literal::String(
        SmallString::from_str(
          "Hello world!"
        )
      )
    )
  ),
  None,
)
```

Codice AJ: Il valore stringa `"Hello world!"` rappresentato con la crate `json_ld`.

3.11. ActivityStreams

ActivityStreams (o *Activity Streams*) é un modello dati per rappresentare attività pianificate, in svolgimento, e svolte, attraverso il formato JSON [108].

É utilizzabile [108], e utilizzato [109] per rappresentare interazioni sociali sul web [110].

É un sottoinsieme di JSON-LD [Sezione 3.10.]: implementare ActivityStreams non richiede necessariamente di implementare tutte le funzionalità di JSON-LD, ma solamente la **compattazione** [Sezione 3.10.2.], **internazionalizzazione** [Sezione 3.10.4.] [Sezione 3.10.5.], e **tipizzazione** [Sezione 3.10.6.] [108].

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "type": "Note",
  "content": "I'm writing my thesis!"
}
```

Codice AK: Documento ActivityStreams rappresentate una breve nota.

Definisce numerose proprietà, e tipi che ne fanno uso aventi relazioni di ereditarietà anche multipla e composizione tra loro [111].

```
{
  "@context": "https://www.w3.org/ns/activitystreams",
  "summary": "Sally accepted an invitation to a party",
  "type": "Accept",
  "actor": {
    "type": "Person",
    "name": "Sally"
  },
  "object": {
    "type": "Invite",
    "actor": "http://john.example.org",
    "object": {
      "type": "Event",
      "name": "Going-Away Party for Jim"
    }
  }
}
```

Codice AL: Documento ActivityStreams che descrive l'accettazione di un invito a una festa attraverso un oggetto di tipo `Accept`, che eredita le proprietà `actor` e `object` da `Activity`, i cui valori sono altri oggetti di tipo rispettivamente `Person` e `Invite`, l'ultimo dei quali contiene ancora a sua volta un oggetto `Event` come valore della sua proprietà `object`.

3.11.1. Crate `activitystreams`

La crate Rust `activitystreams`, utilizzata dal link aggregator Lemmy [112], [113], [114], dà la possibilità di fare uso dei tipi di ActivityStreams all'interno di programmi Rust [115], senza fare uso di JSON-LD [Sezione 3.10.].

Essa definisce per ciascun tipo di ActivityStreams una propria `struct` [115].

Ognuna di queste `struct` é composta internamente una sotto-`struct` per ogni tipo da cui il tipo rappresentato eredita proprietà, più una per sé stessa, e un marcatore usato per serializzare il nome del tipo stesso.

Prendendo come esempio il tipo ActivityStreams `Accept`, che eredita da `Activity`, che eredita da `Object`, avremo una `struct Accept` composta da `AcceptProperties`, `ActivityProperties`, `ObjectProperties`, e il marcatore `AcceptType` [116].

```
pub struct Accept {
    pub kind: AcceptType,
    pub accept_props: AcceptProperties,
    pub object_props: ObjectProperties,
    pub activity_props: ActivityProperties,
}
```

Codice AM: Definizione semplificata della `struct Accept`.

Per implementare serializzazione e deserializzazione di questi tipi, la crate fa poi uso della crate `serde` [Sezione 3.1.15.], combinata con delle macro proprie che implementano tratti e metodi per rendere più facile l'interazione con le proprietà [115].

```
fn get_summary_xsd_string(&self) -> Option<XsdString>;

fn get_summary_rdf_lang_string(&self) -> Option<RdfLangString>;

fn get_many_summary_xsd_strings(&self) -> Option<Vec<&XsdString>>;

fn get_many_summary_rdf_lang_strings(&self) ->
Option<Vec<&RdfLangString>>;
```

Codice AN: Metodi generati su `Accept` per accedere in lettura alla proprietà `summary`. Il primo la legge come una stringa singola, il secondo la legge come una stringa singola con `@language`, il terzo la legge come un vettore di stringhe, e il quarto la legge come un vettore di stringhe con `@language`.

4. Lavoro svolto

4.1. Selezione del linguaggio di programmazione

Per realizzare le astrazioni proposte [Sezione 2.], si é scelto di utilizzare il linguaggio di programmazione Rust [Sezione 3.1.], dato che era già in uso in alcuni componenti di Next-Pyter [117], avente il potenziale di raggiungere elevata efficienza essendo un linguaggio compilato, ed avendo ottima integrazione con gli strumenti per lo sviluppo.

4.2. Creazione del workspace

Si é pensato che per favorire il riutilizzo del codice, sarebbe stato utile realizzare tante piccole librerie separate, ciascuna per ogni specifica trattata.

Si é allora scelto di **creare un workspace Cargo** [Sezione 3.1.10.] all'interno delle quali si sarebbero inserite tutte le librerie realizzate, ottenendo così una gestione centralizzata di tutto il lavoro svolto.

```
[workspace]
resolver = "2"
members = ["acrate_database", "acrate_rd", "acrate_nodeinfo",
"acrate_rdserver", "acrate_activitystreams", "acrate_utils",
"acrate_jsonld_macros", "acrate_jsonld"]
```

Codice AO: Il contenuto finale del file `Cargo.toml` che definisce il workspace di ACRATE.

Allo stesso modo, si é scelto di prefissare il messaggio di ogni commit Git effettuato con il nome delle librerie che esso sarebbe andato a intaccare, in modo che leggendo il registro dei commit sia possibile immediatamente capire di cosa tratta la modifica.

```
`rdserver`: Add missing semicolon
```

Codice AP: Esempio di commit message prefissato con il nome della libreria che esso riguarda.

.idea	jsonld, jsonld_macros, activitystreams: Pieces finally start to fit together	3 weeks ago
.media	First commit	7 months ago
acrate_activitystreams	activitystreams: Complete documentation	2 weeks ago
acrate_database	rdserver: Polish and document	2 weeks ago
acrate_docker	docker: Rename ACRATE_WEBFINGER_BIND_ADDRESS to ACRATE_RDSERVER_BIND_ADDRESS	6 months ago
acrate_jsonld	*: Cleanup	2 weeks ago
acrate_jsonld_macros	*: Cleanup	2 weeks ago
acrate_nodeinfo	*: Cleanup	2 weeks ago
acrate_rd	rd: Fix doctest URL	last week
acrate_rdserver	rdserver: Polish and document	2 weeks ago
acrate_thesis	thesis: Fix typo and describe workspaces in Cargo	19 minutes ago
acrate_utils	utils: Document and polish	2 weeks ago
.editorconfig	*: Add *.rs exception to editorconfig	2 weeks ago
.gitignore	*: Add directory metadata to ignore	2 months ago
Cargo.toml	apub_inbox: Remove, as it's just a stub	2 weeks ago
LICENSE.txt	database: Add crate metadata	7 months ago
README.md	Add apub_inbox to the README	6 months ago

Figura C: Screenshot della root del repository Git di ACRATE, catturato sull'istanza Forgejo in cui il repository é salvato il 2025-06-23.

Stefano Pigozzi	83a36cb882	jsonld: Expand on id capabilities
Stefano Pigozzi	d32233b020	jsonld: Restore implementation of LDConvertSingle for Id
Stefano Pigozzi	4ed8d9dd54	activitystreams: Make Entity inherit from Blank and remove its IRI
Stefano Pigozzi	4ad620edb9	activitystreams: Create Blank vocab
Stefano Pigozzi	3f8bce3e75	activitystreams: Add activitystreams vocabulary
Stefano Pigozzi	1cce96f484	activitystreams: Add thiserror dependency
Stefano Pigozzi	030e49efb8	jsonld_macros: Add support for multiple iris and inherits
Stefano Pigozzi	95a744f5cc	jsonld: Add a method to get the id of a LDSource
Stefano Pigozzi	f0999b34f0	jsonld: Remove type parameters where unneeded
Stefano Pigozzi	2d3dee8f0c	activitystreams: Re-enable href

Figura D: Screenshot parziale del commit log di ACRATE, catturato sull'istanza Forgejo in cui il repository é salvato.

4.3. Implementazione dei resource descriptor in Rust

Per prima cosa, si é scelto di creare un'implementazione Rust dei resource descriptor [Sezione 3.8.], in quanto molte altre tecnologie per l'interoperabilità si basano su di essi per scoprire informazioni su che tipo di interoperabilità sia effettivamente possibile.

Si é quindi creata una crate all'interno del workspace ACRATE con il nome di `acrate_rd`, dove «`rd`» sta per «resource descriptor».

Visto che i resource descriptor possono essere sia in XML [Sezione 3.4.] sia in JSON [Sezione 3.5.], é stato necessario trovare un modo per elaborare entrambi i formati e anche convertire tra essi: la crate `serde` é sembrata ottima allo scopo, in quanto attraverso essa le crate `quick-xml` [Sezione 3.4.1.] e `serde_json` [Sezione 3.5.1.] forniscono capacità di serializzazione e deserializzazione per i rispettivi formati.

Purtroppo, la configurazione speciale necessaria a far funzionare `quick-xml` rende necessaria la definizione di `struct` separate da quelle utilizzate per `serde_json`; si é allora scelto di dividere la crate in tre moduli:

- `jrd`, contenente gli item usati per `serde_json`;
- `xrd`, contenente gli item per `quick-xml`;
- `any`, contenente item in comune o di disambiguazione tra i due moduli precedenti.

4.3.1. Implementazione dei JSON resource descriptor

L'implementazione dei JSON resource descriptor fa un uso convenzionale di `serde`: utilizza le derivazioni macro procedurali `Serialize` e `Deserialize`, l'attributo `#[serde(default)]` per inizializzare `Vec` e `HashMap` nel caso le rispettive chiavi siano assenti o nulle, e l'attributo `#[serde(skip_serializing_if = "Option::is_none")]` per fare in modo che determinate chiavi vengano omesse invece che essere impostate a `null`.

```rust
#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct ResourceDescriptorJRD {
    /// The resource this document refers to.
    #[serde(skip_serializing_if = "Option::is_none")]
    pub subject: Option<String>,

    /// Other names the resource described by this document can be
referred to.
    #[serde(default)]
    pub aliases: Vec<String>,

    /// Additional information about the resource described by this
document.
    #[serde(default)]
    pub properties: HashMap<String, Option<String>>,

    /// Links established between the [`Self::subject`] and other
resources.
    #[serde(default)]
    pub links: Vec<ResourceDescriptorLinkJRD>,
}
```

Codice AQ: Estratto della definizione della `struct` utilizzata per serializzare e deserializzare l'oggetto JSON alla radice dei JSON resource descriptors.

Sono inoltre inclusi metodi per permettere la conversione da item di `acrate_rd::xrd` a item di `acrate_rd::jrd`, ottenendo la possibilità di convertire tra i due formati.

```
impl From<ResourceDescriptorXRD> for ResourceDescriptorJRD {
    fn from(value: ResourceDescriptorXRD) -> Self {
        Self {
            subject: value.subject,
            aliases: value.aliases,
            properties:
value.properties.into_iter().map(From::from).collect(),
            links: value.links.into_iter().map(From::from).collect(),
        }
    }
}
```

Codice AR: Implementazione del tratto di conversione `From` che converte la `struct` di `acrate_rd::xrd` descrivente la radice di un resource descriptor alla `struct` equivalente di `acrate_rd::jrd`. Mentre le proprietà `subject` e `aliases` vengono spostate invariate, `properties` viene convertita da un `Vec` a una `HashMap`, e `links`, un vettore di altre `struct` di `acrate_rd::xrd`, viene convertito ricorsivamente a un vettore di `struct` di `acrate_rd::jrd`.

Si é poi incluso un metodo statico per l'oggetto radice per permettere con una singola chiamata il recupero e la deserializzazione di un JSON resource descriptor collocato a un dato URL.

```
pub async fn get(client: &reqwest::Client, url: reqwest::Url) ->
Result<Self, GetJRDError> {
    use GetJRDError::*;

    log::debug!("Getting JRD document at: {url}");

    log::trace!("Building request...");
    let request = {
        log::trace!("Creating new request...");
        let mut request = reqwest::Request::new(reqwest::Method::GET,
url);

        log::trace!("Setting request headers...");
        let headers = request.headers_mut();

        log::trace!("Setting `Accept: application/jrd+json, application/
json`...");
        let _ = headers.insert(
            reqwest::header::ACCEPT,
            "application/jrd+json, application/json".parse().unwrap(),
        );

        request
    };

  // ...
}
```

Codice AS: Breve estratto della funzione `ResourceDescriptorJRD::get` *che permette il recupero di un JSON resource descriptor remoto.*

Infine, é presente un `enum` rappresentante gli errori che possono avvenire durante quel recupero, su cui é implementato il `trait` `std::error::Error` attraverso la crate `thiserror` [Sezione 3.1.16.].

```
#[derive(Debug, Error)]
pub enum GetJRDError {
    #[error("the HTTP request failed")]
    Request(reqwest::Error),

    #[error("the Content-Type header of the response is missing")]
    ContentTypeMissing,

    #[error("the Content-Type header of the response cannot be converted
to a &str")]
    ContentTypeUnprintable(reqwest::header::ToStrError),

    #[error("the Content-Type header of the response is not a valid
media type")]
    ContentTypeInvalid(mediatype::MediaTypeError),

    #[error("the Content-Type header of the response is not a supported
media type")]
    ContentTypeUnsupported,

    #[error("the document failed to be parsed as JSON")]
    Parse(reqwest::Error),
}
```

Codice AT: Estratto della definizione dell'enum risultante da un errore di recupero di un JSON resource descriptor.

4.3.2. Implementazione degli XML resource descriptor

L'implementazione degli XML resource descriptor segue la stessa struttura di quella usata per implementare i JSON resource descriptor, con alcune differenze chiave.

Gli XML resource descriptor usano nomi *CamelCase* per i loro nodi; é stato quindi necessario usare l'attributo `#[serde(rename = "...")]` per associarli i campi a identificatori diversi da quelli che hanno nel codice.

```
#[derive(Debug, Clone, Serialize, Deserialize)]
#[serde(rename = "XRD")]
pub struct ResourceDescriptorXRD {
    #[serde(rename = "Subject")]
    #[serde(skip_serializing_if = "Option::is_none")]
    pub subject: Option<String>,

    #[serde(rename = "Alias")]
    #[serde(default)]
    pub aliases: Vec<String>,

    #[serde(rename = "Property")]
    #[serde(default)]
    pub properties: Vec<ResourceDescriptorPropertyXRD>,

    #[serde(rename = "Link")]
    #[serde(default)]
    pub links: Vec<ResourceDescriptorLinkXRD>,
}
```

Codice AU: Estratto della definizione della `struct` rappresentante la radice di un XML resource descriptor.

Alcuni dei campi vanno processati come proprietà XML dei nodi in cui si trovano: per indicare questo, `quick-xml` richede che essi siano rinominati con identificatori prefissati dal carattere `@`, risultando nell'attributo `#[serde(rename = "@...")]`, che si é utilizzato in alcune `struct` del modulo `xrd`.

```
#[derive(Debug, Clone, Serialize, Deserialize)]
#[serde(rename = "Link")]
pub struct ResourceDescriptorLinkXRD {
    #[serde(rename = "@rel")]
    pub rel: String,

    #[serde(rename = "@type")]
    #[serde(skip_serializing_if = "Option::is_none")]
    pub r#type: Option<MediaTypeBuf>,

    #[serde(rename = "@href")]
    #[serde(skip_serializing_if = "Option::is_none")]
    pub href: Option<String>,

    #[serde(default)]
    #[serde(rename = "Title")]
    pub titles: Vec<ResourceDescriptorTitleXRD>,

    #[serde(default)]
    #[serde(rename = "Property")]
    pub properties: Vec<ResourceDescriptorPropertyXRD>,

    #[serde(rename = "@template")]
    #[serde(skip_serializing_if = "Option::is_none")]
    pub template: Option<String>,
}
```

Codice AV: Estratto della definizione della `struct` rappresentante il nodo interno `<Link/>` di un XML resource descriptor.

Infine, alcuni campi vanno processati come il contenuto di nodi XML: per indicarlo, é previsto che il loro identificatore venga impostato a `$text`, usando l'attributo `#[serde(rename = "$text"]`.

```rust
/// A title of the resource put in relation.
#[derive(Debug, Clone, Serialize, Deserialize)]
#[serde(rename = "Title")]
pub struct ResourceDescriptorTitleXRD {
    /// The language of the title.
    #[serde(rename = "@lang")]
    pub language: String,

    /// The title itself.
    #[serde(rename = "$text")]
    pub value: String,
}
```

Codice AW: Estratto della definizione della `struct` rappresentante il nodo interno `<Title>...</Title>` di un XML resource descriptor.

4.3.3. Implementazione delle astrazioni comuni tra i due formati

L'item principale del modulo `any` é `ResourceDescriptor`, un `enum` di disambiguazione tra `ResourceDescriptorJRD` e `ResourceDescriptorXRD`.

```rust
/// A resource descriptor in any format.
#[derive(Debug, Clone)]
pub enum ResourceDescriptor {
    /// The resource descriptor is in JRD format.
    JRD(ResourceDescriptorJRD),
    /// The resource descriptor is in XRD format.
    XRD(ResourceDescriptorXRD),
}
```

Codice AX: La definizione di `ResourceDescriptor`.

Su di esso, si sono implementati due metodi:
- `get`, che tenta il recupero di un resource descriptor ad un dato URL, prima provando a recuperare un XML resource descriptor, poi un JSON resource descriptor, poi un JSON resource descriptor aggiungendo `.json` alla fine dell'URL ricevuto;
- `discover_hostmeta`, che prova a recuperare il resource descriptor «host-meta» all'URL noto `/.well-known/host-meta`.

Si sono inoltre incluse due strutture rappresentanti gli errori potenzialmente risultanti da chiamate dei due metodi appena descritti, anch'esse facenti uso di `thiserror::Error` [Sezione 3.1.16.].

4.4. Creazione servizio web di generazione resource descriptor

Al fine di rendere immediatamente fruibili le funzionalità offerte da `acrate_rd` [Sezione 4.3.], si é realizzata la crate `acrate_rdserver`, un piccolo servizio web [Sezione 3.2.] che risponde a tutte le richieste leggendo informazioni da una base di dati e restituendo un resource descriptor con le informazioni lette.

Si é scelto di utilizzare axum [Sezione 3.2.4.] per la sua realizzazione, sfruttando appieno la sua versatilità per realizzare una base efficace e al tempo stesso facilmente estendibile qualora la si volesse riutilizzare per progetti diversi, utilizzando minijinja [Sezione 3.2.8.] ovunque fosse necessario l'output di pagine HTML in modo da tenere il codice più organizzato, e facendo uso di log [Sezione 3.1.17.] e pretty_env_logger [Sezione 3.1.18.] per il logging.

4.4.1. Realizzazione di funzioni di utilità per servizi web

Inizialmente per questa tesi era ponderata la possibilità di inclusione di più servizi web dello stesso tipo di acrate_rdserver, quindi si é astratto l'utilizzo di axum, minijinja, log, e pretty_env_logger a una crate separata, che si é chiamata acrate_utils perché contenente «funzioni di utilità miste».

É composta da un modulo principale e tre sotto-moduli:

- acrate_utils stessa, contenente la funzione macro dichiarativa [Sezione 3.1.13.] web_server! usata per effettuare l'inizializzazione del servizio web;
- acrate_utils::ext, contenente alias abbreviati per gli estrattori axum personalizzati [Sezione 3.2.6.] di uso comune in ACRATE, nello specifico una Extension contenente un minijinja::Environment, abbreviata a ExtMj;
- acrate_utils::init, contenente le funzioni di inizializzazione di tutte le funzionalità di logging, socket listening, template formatting, e resource routing necessarie;
- acrate_utils::run, contenente una funzione per avviare il servizio web una volta configurato e gestire gli errori verificatisi in esso.

```
web_server!(
    on: *config::ACRATE_RDSERVER_BIND_ADDRESS(),
    templates: [
        "rd.html.j2"
    ],
    routes: {
        "/{*path}"        => get(routes::webfinger_handler),
        "/.healthcheck" => get(routes::healthcheck_handler)
    }
);
```

Codice AY: Esempio di chiamata alla funzione macro dichiarativa web_server! estratto da acrate_rdserver, che mostra la sintassi richiesta per specificarne i parametri. Il «parametro» on: deve corrispondere all'indirizzo del socket su cui il server deve essere in ascolto per nuove richieste, il «parametro» templates: deve corrispondere a una serie di stringhe corrispondenti ai percorsi dei file che devono essere inclusi nel minijinja::Environment relativi alla posizione del file di codice in cui si trova la chiamata, e il «parametro» routes: deve corrispondere a una serie di coppie risorsa–handler-richiesta separate da =>.

```
/// Setup an [`axum`] webserver on the given socket with the given
[`minijinja`] templates and [`axum`] routes.
#[macro_export]
macro_rules! web_server {
    (
    on: $socket_addr:expr,
    templates: [
      $( $template_path:literal ),*
    ],
    routes: {
      $( $path:literal => $route:expr ),*
    }
  ) => {
    $crate::init::init_logging();
    let listener = $crate::init::init_listener($socket_addr).await;
    let mut mj = $crate::init::init_minijinja();
    $(
      $crate::add_minijinja_template!(mj, $template_path);
    )*
    let router = $crate::init::init_router();
    $(
      $crate::add_axum_route!(router, $path, $route);
    )*
    let router = $crate::ext::install_minijinja(router, mj);
    $crate::run::run_server(listener, router).await
  };
}
```

Codice AZ: Definizione della funzione macro dichiarativa `web_server!`.

4.4.2. Connessione con basi di dati

Per lo stesso motivo, si é astratta la funzionalità di accesso a basi di dati in una ulteriore crate, `acrate_database`.

Essa fa uso di `diesel` [Sezione 3.6.1.] e `diesel_async` [Sezione 3.6.2.] per permettere l'interazione con un database PostgreSQL [Sezione 3.6.] all'interno di codice Rust.

Usando `diesel_cli` si é creata una serie di migrazioni che creano le tabelle necessarie al funzionamento di `acrate_rdserver`, che sono state collocate in `./acrate_database/migrations`.

```sql
CREATE TABLE IF NOT EXISTS meta_links (
  id UUID DEFAULT gen_random_uuid(),
  document BPCHAR NOT NULL,
  pattern BPCHAR NOT NULL,
  rel BPCHAR NOT NULL,
  type BPCHAR,
  href BPCHAR,
  template BPCHAR,

  CONSTRAINT either_href_or_template_not_null CHECK (
    (href IS NOT NULL AND template IS NULL)
    OR
    (href IS NULL AND template IS NOT NULL)
  ),
  PRIMARY KEY (id)
);
```

Codice BA: Esempio della creazione di una tabella per `acrate_rdserver` tratto da una migrazione. Essa rappresenta un nodo `<Link/>` di un resource descriptor. I tipi stringa in essa usano `BPCHAR` per ignorare gli spazi terminanti. Si é fatto uso di `CONSTRAINT` complessi per verificare alcuni dei vincoli imposti dalla specifica dei resource descriptor: in questo caso, si sta verificando che solo uno tra `href` e `template` non sia NULL. Il prefisso `meta` del nome della tabella deriva da `host-meta`, il nome del file contenente il resource descriptor per un dominio.

Basandosi su esse, `diesel_cli` ha generato una serie di macro per utilizzo interno, che ha collocato in `acrate_database::schema`.

```rust
diesel::table! {
    meta_links (id) {
        id -> Uuid,
        document -> Bpchar,
        pattern -> Bpchar,
        rel -> Bpchar,
        #[sql_name = "type"]
        type_ -> Nullable<Bpchar>,
        href -> Nullable<Bpchar>,
        template -> Nullable<Bpchar>,
    }
}
```

Codice BB: La stessa tabella creata in [Codice BA], rappresentata da una macro ad utilizzo interno di `diesel`.

In `acrate_database::meta` si sono poi costruite una coppia di `struct` per ciascuna tabella: una rappresentate i valori immagazzinati nel database, e una rappresentante i valori passati al database per la creazione di un nuovo record.

```
#[derive(Debug, Queryable, QueryableByName, Identifiable, Selectable,
Associations)]
#[diesel(belongs_to(MetaLink))]
#[diesel(table_name = schema::meta_link_properties)]
#[diesel(check_for_backend(Pg))]
pub struct MetaLinkProperty {
    pub id: Uuid,
    pub meta_link_id: Uuid,
    pub rel: String,
    pub value: Option<String>,
}
```

Codice BC: Estratto di codice della struct «in lettura» corrispondente alla tabella `meta_link_properties`. Si può vedere che é presente il parametro `id`, e che implementa fra le altre cose `Queryable`.

```
#[derive(Debug, Insertable)]
#[diesel(belongs_to(MetaLink))]
#[diesel(table_name = schema::meta_link_properties)]
#[diesel(check_for_backend(Pg))]
pub struct MetaLinkPropertyInsert {
    pub meta_link_id: Uuid,
    pub rel: String,
    pub value: Option<String>,
}
```

Codice BD: Estratto di codice della struct «in scrittura» della stessa tabella di [Codice BC]. Si nota che é assente il parametro `id`, in quanto selezionato da PostgreSQL al momento della creazione del record, e che implementa `Insertable`.

Si sono infine implementati sui tipi «in lettura» dei metodi statici che permettono di effettuare query sincrone (`query_`) o asincrone (`aquery_`) per ottenere valori di quel tipo, e sui tipi «in scrittura» dei metodi che ne permettono l'inserimento e quindi la conversione a un tipo «in lettura» (`to_inserted`).

Tabella	Colonne	Descrizione
`meta_subjects`	`id`, `document`, `pattern`, `subject`, `redirect`	Soggetto con resource descriptor.
`meta_aliases`	`id`, `document`, `pattern`, `aliases`	Nodo `<Alias/>` di un resource descriptor.
`meta_links`	`id`, `document`, `pattern`, `rel`, `type`, `href`, `template`	Nodo `<Link/>` di un resource descriptor.
`meta_link_properties`	`id`, `meta_link_id`, `rel`, `value`	Nodo `<Property/>` di un nodo `<Link/>`.
`meta_link_titles`	`id`, `meta_link_id`, `language`, `value`	Nodo `<Title/>` di un nodo `<Link/>`.
`meta_properties`	`id`, `document`, `pattern`, `rel`, `value`	Nodo `<Property/>` di un resource descriptor.

Tabella A: Schema delle tabelle utilizzate da `acrate_rdserver`. Più informazioni su come esse vengono interpretate sono fornite in [Sezione 4.4.7.].

4.4.3. Containerizzazione

Ancora, aspettandocisi di realizzare più servizi web, si sono create le istruzioni per creare immagini Docker [Sezione 3.7.1.] di ciascun servizio web di ACRATE, e un progetto Docker Compose [Sezione 3.7.3.] che avvii tutti i servizi web e le loro dipendenze esterne, collocandolo in `/acrate_docker` come se fosse una crate del workspace Rust.

Il `Dockerfile` usa un'architettura multi-stadio per minimizzare la dimensione delle immagini risultanti: prima usa l'immagine base `rust` [80] per compilare tutto il progetto, poi prende l'immagine `rust:slim`, vi aggiunge la libreria `libpq5` per consentire interazioni con PostgreSQL [Sezione 3.6.], e vi copia l'interno l'eseguibile compilato del servizio.

Fa uso dell'«additional context» `source`, che deve essere impostato al momento della creazione dell'immagine alla root del repository di ACRATE, o attraverso `docker buildx --build-context source=/la/mia/directory` o attraverso la chiave `additional_contexts: ["source=/la/mia/directory"]` di un progetto Docker Compose, necessario per permettere al builder di accedere a una directory superiore a quella in cui il builder viene chiamato o in cui si trova il progetto Docker Compose.

```
FROM rust AS base_builder
WORKDIR /usr/src/acrate
COPY --from=source ./acrate_database ./acrate_database
COPY --from=source ./acrate_rd ./acrate_rd
COPY --from=source ./acrate_rdserver ./acrate_rdserver
COPY --from=source ./Cargo.toml ./Cargo.toml
COPY --from=source ./Cargo.lock ./Cargo.lock

FROM rust:slim AS base_runner
RUN apt-get update
RUN apt-get upgrade --assume-yes
RUN apt-get install --assume-yes libpq5
WORKDIR /usr/local/bin
ENV RUST_LOG="warn"

FROM base_builder AS migrate_build
RUN cargo build --release --package=acrate_database --features=bin --
bin=acrate_database_migrate

FROM base_runner AS migrate
COPY --from=migrate_build /usr/src/acrate/target/release/
acrate_database_migrate /usr/local/bin/acrate_database_migrate
ENTRYPOINT ["acrate_database_migrate"]
ENV RUST_LOG="warn,acrate_database_migrate=info"

FROM base_builder AS rdserver_build
RUN cargo build --release --package=acrate_rdserver --
bin=acrate_rdserver

FROM base_runner AS rdserver
COPY --from=rdserver_build /usr/src/acrate/target/release/
acrate_rdserver /usr/local/bin/acrate_rdserver
ENTRYPOINT ["acrate_rdserver"]
HEALTHCHECK CMD ["curl", "http://127.0.0.1/.healthcheck"]
ENV RUST_LOG="warn,acrate_rdserver=info"
```

Codice BE: Il Dockerfile completo di `acrate_docker`. Si può osservare che le prime istruzioni `COPY` fanno uso del parametro `--from=source`, nonostante non sia definita un'immagine `source`.

Similarmente, il progetto Docker Compose usa anch'esso un'architettura componibile, ed é diviso in due file, `compose.yml` e `debug.compose.yml`; il primo pensato per un setup «di produzione» dei servizi web, e il secondo che, una volta unito al primo attraverso l'opzione `--file` di Docker Compose, gli aggiunge funzionalità utili allo sviluppo (attualmente solo esponendo i servizi direttamente su `localhost`, bypassando il reverse proxy in ingresso).

```yaml
services:
  rdserver:
    build:
      dockerfile: "./Dockerfile"
      additional_contexts:
      - "source=.."
      target: "rdserver"
    restart: "unless-stopped"
    environment:
      ACRATE_RDSERVER_BIND_ADDRESS: "0.0.0.0:80"
      ACRATE_DATABASE_URL: *database_url
    expose:
    - 80
    depends_on:
      database:
        condition: "service_healthy"
      migrate:
        condition: "service_completed_successfully"
```

Codice BF: Estratto del progetto Docker Compose `compose.yml`. Si può osservare che la chiave `additional_contexts` é stata impostata a `[source=..]` per permettere l'accesso alla cartella immediatamente superiore, corrispondente alla radice del repository.

É pensato per essere completamente plug-and-play; avendo una copia del repository di ACRATE, é sufficiente entrare nella cartella `acrate_docker` ed eseguire `docker compose up` per avviare tutto ciò che é disponibile.

```sh
cd ./acrate_docker
docker compose up
```

Codice BG: Comando per eseguire il setup «di produzione» di `acrate_docker`.

```sh
cd ./acrate_docker
docker compose --file='compose.yml' --file='debug.compose.yml' up
```

Codice BH: Comando per eseguire il setup «di sviluppo» di `acrate_docker`.

4.4.4. Implementazione configurabilità delle crate

Per rendere l'immagine Docker facilmente configurabile, si é creato un modulo `config` sia in `acrate_database` sia in `acrate_rdserver` stessa che facesse uso della crate `micronfig` [Sezione 3.7.4.] per permettere la configurazione di alcuni parametri attraverso variabili di ambiente, facili da impostare al momento di creazione di un container Docker [Sezione 3.7.2.] o di un progetto Docker Compose [Sezione 3.7.3.].

```
micronfig::config! {
    ACRATE_DATABASE_URL: String,
}
```

Codice BI: Configurazione di `acrate_database`. La variabile `ACRATE_DATABASE_URL` specifica a quale database l'applicazione dovrà connettersi.

```
micronfig::config!(
    ACRATE_RDSERVER_BIND_ADDRESS: String > std::net::SocketAddr,
);
```

Codice BJ: Configurazizone di `acrate_rdserver`. La variabile `ACRATE_RDSERVER_BIND_ADDRESS` specifica il socket a cui l'applicazione dovrà essere in ascolto per la ricezione di richieste HTTP.

Definire parametri di configurazione in una crate libreria garantisce che essi siano configurati con quella specifica chiave, lasciando più libertà all'utilizzatore dell'eseguibile finale di decidere come deduplicare la configurazione; questo potrebbe determinare un problema nel caso due librerie usino la stessa chiave, ma esso é immediatamente risolto prefissando la chiave di configurazione con il nome della libreria stessa, per esempio definendo `ACRATE_RDSERVER_BIND_ADDRESS` invece che `BIND_ADDRESS`.

```
services:
  uno:
    environment: &env
      ACRATE_DATABASE_URL: "postgres:///postgres?
host=database&user=postgres&password=acrate"
      RUST_LOG: "info"
    # ...
  due:
    environment: *env
    # ...
```

Codice BK: Progetto Docker Compose di esempio, in cui i container `uno` e `due` vengono configurati con lo stesso `ACRATE_DATABASE_URL` e `RUST_LOG`, nonostante i loro valori siano scritti una volta sola. Ciò avviene tramite l'anchor YML `env`, dichiarata in `uno` e poi referenziata in `due`. Questo é possibile solo perché entrambi i container usano la stessa chiave per configurare la stessa funzione.

```yaml
services:
  uno:
    env_file: ".env"
    # ...
  due:
    env_file: ".env"
    # ...
```

Codice BL: Progetto Docker Compose di esempio in cui si fa la stessa cosa di [Codice BK] attraverso l'utilizzo dello stesso file `.env` esterno per entrambi i container.

```yaml
services:
  uno:
    environment:
      ACRATE_DATABASE_URL: "postgres:///postgres?
host=database&user=uno&password=uno"
      RUST_LOG: "warn"
    # ...
  due:
    environment:
      ACRATE_DATABASE_URL: "postgres:///postgres?
host=database&user=due&password=due"
      RUST_LOG: "debug"
    # ...
```

Codice BM: Progetto Docker Compose di esempio simile a quello di [Codice BK] in cui nonostante i due eseguibili condividano la stessa chiave di configurazione, essa é configurata a un valore diverso per ciascun container.

4.4.5. Definizione di un healthcheck

Si é scelto di includere una risorsa speciale di «healthcheck» in tutti i servizi web creati al percorso `/.healthcheck`.

Si é fatto in modo che le richieste HTTP `GET` inviate lì vengano gestite da `healthcheck_handler`, che tenta di effettuare una connessione al database e restituisce `204 No Content` o `502 Bad Gateway` rispettivamente se essa riesce o fallisce.

```rust
/// Handler for `/.healthcheck`.
pub async fn healthcheck_handler() -> Result<StatusCode, StatusCode> {
    log::debug!("Handling an healthcheck request!");

    log::trace!("Making sure the database is up...");
    let _conn = connect_async().await.map_err(|_|
StatusCode::BAD_GATEWAY)?;

    log::trace!("Healthcheck successful! Everything's fine!");
    Ok(StatusCode::NO_CONTENT)
}
```

Codice BN: Definizione della funzione `healthcheck_handler`.

Si é poi usata l'istruzione `HEALTHCHECK` nel Dockerfile del servizio, in modo tale che periodicamente Docker usi cURL [Sezione 3.3.] per richiedere quella risorsa, e usi il risultato della richiesta per determinare se il container é funzionale oppure no.

```
HEALTHCHECK CMD ["curl", "http://127.0.0.1/.healthcheck"]
```

Codice BO: L'istruzione Dockerfile utilizzata per configurare l'healthcheck nell'immagine Docker di `acrate_rdserver`.

4.4.6. Configurazione per l'esecuzione automatica delle migrazioni del database

`acrate_database`, oltre alla crate libreria, include una crate eseguibile, `acrate_database_migrate`, che si connette al database nello stesso modo in cui farebbe `acrate_database` e su quel database esegue le migrazioni che non sono ancora state applicate, uscendo con l'exit code `2` se si verifica un errore nel processo, o uscendo con `0` se le migrazioni sono riuscite.

Questo torna utile nella realizzazione di un progetto Docker Compose [Sezione 3.7.3.]: se il cui processo principale (entrypoint) di un container Docker [Sezione 3.7.2.] termina, e svolge un compito la cui riuscita é segnalata da un exit code, é possibile usare quel risultato come requisito perché altri container possano essere avviati.

```yaml
services:
  # ...
  migrate:
    # ...
    depends_on:
      database:
        condition: "service_healthy"

  rdserver:
    # ...
    depends_on:
      database:
        condition: "service_healthy"
      migrate:
        condition: "service_completed_successfully"
```

*Codice BP: Estratto del progetto Docker Compose di sviluppo di ACRATE. Il servizio `migrate`, che esegue `acrate_database_migrate`, non viene avviato fino a quando il servizio `database` non é pronto a ricevere connessioni, mentre il servizio `rdserver`, che esegue `acrate_rdserver`, non viene avviato fino a quando il `database` non é pronto **E** le migrazioni sono state applicate con successo.*

4.4.7. Costruzione dei resource descriptor

Per comporre i resource descriptor da restituire, `acrate_rdserver` effettua varie query al database.

Tutte le query effettuate usano le stesse clausole `WHERE`, richiedendo che:

- la colonna `document` coincida con il percorso relativo della richiesta HTTP ricevuta dal server; **e**
- la colonna `pattern` sia un match `ILIKE` con il valore del parametro `resource` della richiesta HTTP ricevuta dal server.

Si offre così versatilità al servizio che fa uso di `acrate_rdserver`: può ad esempio scegliere di rendere valido un dato record per tutte le resource specificando `%` come `pattern`, di renderlo valido solo per le resource in `example.org` con `https://example.org/%`, o di renderlo valido per una specifica risorsa specificandone il nome esattamente.

La prima query effettuata é alla tabella `meta_subject`: se essa restituisce dei uno o più valori, si prende in considerazione il primo restituito.

Se il subject ha un valore nella colonna `redirect`, l'elaborazione viene immediatamente terminata restituendo `302 Found` e reindirizzando alla pagina con quel valore.

Infine, si effettuano poi query per determinare i nodi interni del resource descriptor, includendo tutti quelli restituiti, alle tabelle:

1. `meta_aliases`
2. `meta_properties`

3. `meta_links`
   1. `meta_link_properties`
   2. `meta_link_titles`

4.4.8. Content negotiation in `acrate_rdserver`

Il servizio web realizzato ha supporto per la content negotiation [Sezione 3.2.3.].

Innanzitutto, si imposta l'header `Vary` di ogni risposta ad `Accept`.

Poi, viene effettuato pattern matching su ogni formato accettato in ordine di priorità, fino a quando non ne viene determinato uno accettabile; se nessuno dei tipi proposti é accettabile, allora viene restituito un errore HTTP `406 Not Acceptable`.

Pattern	Risultato	Riferimento
`*/*`	JSON resource descriptor	[Sezione 4.3.1.]
`application/json`	JSON resource descriptor	[Sezione 4.3.1.]
`application/jrd+json`	JSON resource descriptor	[Sezione 4.3.1.]
`application/xml`	XML resource descriptor	[Sezione 4.3.2.]
`application/xrd+xml`	XML resource descriptor	[Sezione 4.3.2.]
`text/html`	Human-friendly resource descriptor	[Sezione 4.4.9.]

Tabella B: Documento inviato da `acrate_rdserver` in corrispondenza di ciascun formato.

4.4.9. Resource descriptor human-friendly

Una volta implementata la content negotiation per JRD e XRD, si é pensato potesse essere utile avere anche una rappresentazione dei resource descriptor in HTML, in modo che essi potessero essere comprensibili a colpo d'occhio se aperti da un browser.

Usando `minijinja` [Sezione 3.2.8.], si é quindi realizzata la pagina visibile in [Figura E].

Figura E: Esempio di resource descriptor WebFinger ispirato a Mastodon visualizzato in formato human-friendly mentre é in uso un tema colori chiaro sul browser.

4.5. Implementazione di NodeInfo in Rust

Vista la similitudine architetturale con i resource descriptor, si é scelto di realizzare una crate Rust per NodeInfo [Sezione 3.9.], chiamata `acrate_nodeinfo`, utilizzante `serde` [Sezione 3.1.15.] e `serde_json` [Sezione 3.5.1.] per elaborare i documenti.

É composta da un singolo modulo contenente tutte le strutture rilevanti, cioé:

- `NodeInfo`, struttura di disambiguazione tra le versioni 1 e 2 di NodeInfo;
- `NodeInfo1`, radice di un documento versione 1;
  - `NodeInfo1Software`, corrispondente all'oggetto al percorso `.software` del documento JSON;
  - `NodeInfo1Protocols`, corrispondente all'oggetto al percorso `.protocols` del documento JSON;
  - `NodeInfo1Usage`, corrispondente all'oggetto al percorso `.usage` del documento JSON;
    - `NodeInfo1UsageUsers`, corrispondente all'oggetto al percorso `.usage.users` del documento JSON;
- `NodeInfo2`, radice di un documento versione 2, che fa uso delle strutture della versione 1 ove esse non abbiano subito variazioni;
  - `NodeInfo2Instance`, corrispondente all'oggetto al percorso `.instance` del documento JSON.

Come per `acrate_rd::any::ResourceDescriptor`, si é implementato su `NodeInfo1` e `NodeInfo2` un metodo `get` che permette il recupero di un documento NodeInfo remoto a partire da un URL, poi si é implementato un terzo metodo su `NodeInfo`, `get_latest_wellknown`, che, dato un URL puntante a un resource descriptor [Sezione 3.8.], lo recupera usando `acrate_rd`, poi cerca in esso un `<Link/>` con un valore di `rel` corrispondente a quello delle versioni di NodeInfo supportate `http://nodeinfo.diaspora.software/ns/schema/*.*`, preferendo versioni maggiori quando possibile, e lo usa per recuperare il documento con il metodo `get` appropriato.

4.6. Definizione di astrazioni per interagire con JSON-LD

Dopo aver implementato resource descriptor e NodeInfo, tecnologie di discoverability, che permettono a servizi web di scoprire l'esistenza di un l'altro, si é pensato potesse essere utile procedere con l'implementazione di una tecnologia di interscambio dati.

Pensando ad ActivityStreams [Sezione 3.11.] come ipotetico formato dati futuro, si é scelto di realizzare una crate Rust per JSON-LD [Sezione 3.10.], il formato dati su cui esso si poggia.

4.6.1. Stato dell'arte

Prima di iniziare con lo sviluppo, si sono esaminate le crate pre-esistenti trattanti l'argomento: `serde_json`, `json_ld` e `activitystreams`.

`serde_json` [Sezione 3.5.1.] é facile da utilizzare, ma non é dotata degli strumenti necessari per trattare le particolarità di JSON-LD, come le molteplicità di forme che uno stesso documento può avere, o meccanismi per effettuare la convalida dei tipi al momento del parsing.

`json_ld` [Sezione 3.10.7.] é in grado di processare ottimalmente JSON-LD, ma é molto verbosa da scrivere, e dato un oggetto di un tipo, non permette di scoprirne le proprietà disponibili, in quanto gli oggetti sono trattati similarmente a come sarebbero trattate delle `HashMap`.

`activitystreams` [Sezione 3.11.1.] implementa il minimo di JSON-LD indispensabile per processare tipi ActivityStreams, ma il mancato utilizzo dell'algoritmo di espansione JSON-LD rende necessario l'utilizzo di diversi metodi per avere interoperabilità completa.

Visto ciò, si é deciso di realizzare una propria crate, `acrate_jsonld`, che soddisfacesse i requisiti desiderati.

Dato che la crate `json_ld` é già dotata degli algoritmi necessari per l'elaborazione di JSON-LD, si é scelto di usarla come base, almeno inizialmente, costruendovi sopra delle interfacce per permetterne un utilizzo più facile e scopribile da editor.

4.6.2. Requisiti dell'astrazione

Si sono proposti i seguenti requisiti per qualsiasi astrazione si sarebbe realizzata:

1. a ciascun tipo JSON-LD si desideri utilizzare deve essere associato un tipo Rust;

2. deve essere possibile avere relazioni di ereditarietà anche multipla per le proprietà dei tipi, in modo da poter implementare specifiche come ActivityStreams che descrivono i tipi in quel modo [Sezione 3.11.];
3. deve essere possibile richiedere che ad una funzione sia passato un tipo Rust avente un determinato tipo JSON-LD;
4. deve essere possibile richiedere che ad una funzione sia passato un tipo Rust *avente le proprietà*, o *ereditante*, un dato tipo JSON-LD;
5. si devono ricevere solo suggerimenti rilevanti per accedere alle proprietà del tipo JSON-LD, come in [Figura F], e non suggerimenti relativi all'implementazione interna come in [Figura G];
6. non deve perdere informazioni già presenti nell'oggetto JSON-LD durante l'elaborazione come tipo Rust, anche se queste informazioni non sono previste dal tipo JSON-LD, in quanto potrebbero essere richieste da un software difettoso o che estende la specifica;
7. deve essere agnostica alla crate utilizzata per elaborare internamente i dati, in modo da poter facilmente sostituire la crate `json_ld` qualora risulti problematica durante lo sviluppo.

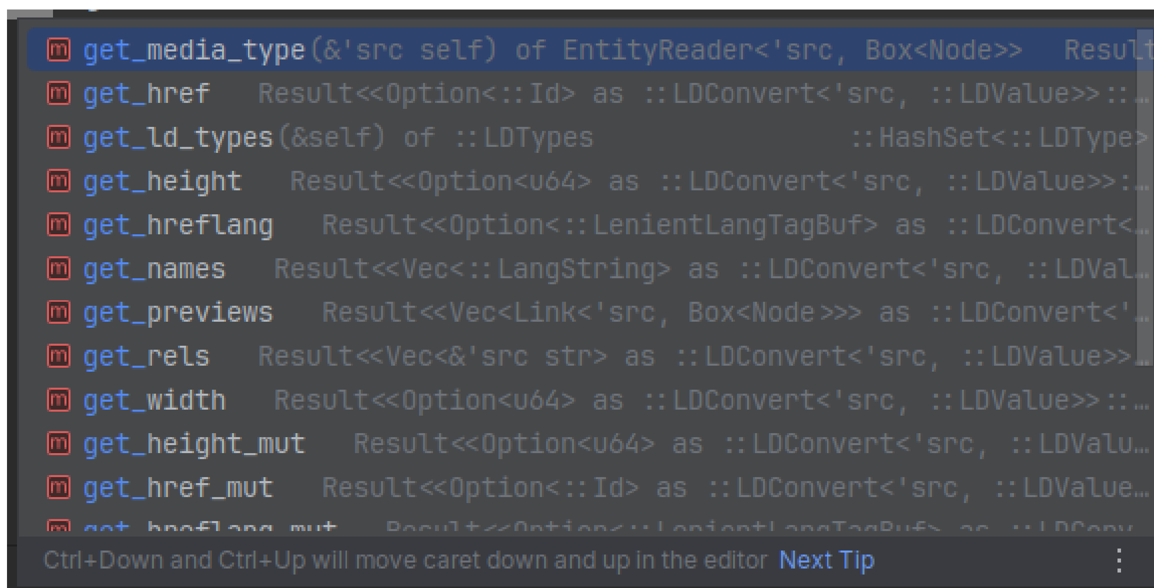

Figura F: Screenshot degli autocompletamenti suggeriti da IntelliJ nella versione finale di `acrate_activitystreams` *come metodi con un prefisso di* `get_` *di* `acrate_activitystreams::vocabs::Link`*. L'effetto dei metodi é immediatamente chiaro, e non ci sono suggerimenti superflui, come quelli di* `Box<json_ld::Node>`*, non necessari all'utilizzo di* `Link`*.*

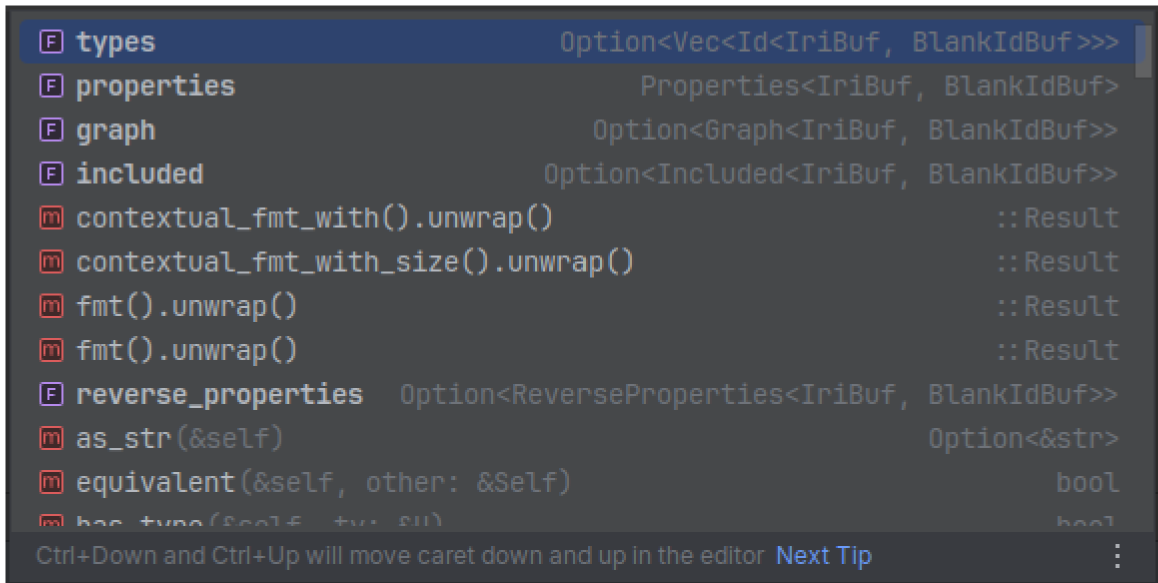

Figura G: Screenshot degli autocompletamenti suggeriti da IntelliJ come metodi di `Box<json_ld::Node>`.

4.6.3. Implementazione dell'astrazione

Determinare un modo pratico per soddisfare tutti i requisiti dell'astrazione non é stato facile, soprattutto rispettando le severe limitazioni imposte dalla orphan rule [Sezione 3.1.8.], e ha richiesto varie iterazioni e riscritture del codice.

I requisiti 1, 2, 3 e 4 si prestano molto bene all'utilizzo di `trait` [Sezione 3.1.6.] vincolati come in [Codice BQ], ma ciò é in diretto contrasto con il requisito 5, in quanto implementare `trait` può solo *aggiungere* metodi ad un tipo concreto.

```rust
trait OrderedCollectionPage: OrderedCollection + CollectionPage {
  // ...
}
```

Codice BQ: Esempio di come si sarebbe potuta realizzare ereditarietà multipla vincolando il `trait` `OrderedCollectionPage` *a poter essere implementato solo se* `OrderedCollection` *e* `CollectionPage` *fossero stati implementati sull'implementatore di* `OrderedCollectionPage`.

Un'idea é stata quella di rappresentare tutto come `struct`, e implementare `trait` di conversione come `From` da un tipo all'altro come in [Codice BR], ma ciò é in contrasto con il requisito 6, in quanto questo risulterebbe in una perdita di informazione; stessa cosa facendo uso di composizione, come fa la crate `activitystreams` [Sezione 3.11.1.].

```
struct OrderedCollectionPage {
    // proprietà proprie di OrderedCollectionPage
    start_index: Option<usize>,
    // proprietà di OrdreredCollection
    total_items: usize,
    // proprietà di CollectionPage
    next: String,
}
```

Codice BR: Esempio di come si sarebbe potuta realizzare una `struct` con tutte le proprietà del tipo JSON-LD rappresentato.

Un modo comune con cui le restrizioni imposte dalla orphan rule [Sezione 3.1.8.] vengono evitata é attraverso il pattern «newtype»: creare una nuova `struct` contenente un unico campo del tipo a cui verrà effettuato l'accesso, rendendo il tipo definito nella crate alla quale si sta lavorando e quindi senza limitazioni imposte.

Così facendo, si é ottenuto un nuovo tipo, soddisfacendo i requisiti 1 e 3, non intaccando le informazioni già presenti nel tipo contenuto, ottenendo il requisito 6.

```
struct OrderedCollectionPage(
    pub Box<json_ld::Node>
)
```

Codice BS: Esempio del pattern newtype in cui il nuovo tipo ha la proprietà dell'oggetto contenuto.

L'accesso alle proprietà può essere implementato attraverso metodi sul nuovo tipo, soddisfacendo il requisito 5.

```
impl OrderedCollectionPage {
    fn get_start_index(&self) -> Option<usize> { /* ... */ }
    fn get_total_items(&self) -> Option<usize> { /* ... */ }
    fn get_next(&self) -> Option<usize> { /* ... */ }
}
```

Codice BT: Esempio di implementazione metodi di accesso alle proprietà direttamente sul newtype.

Per realizzare i requisiti 2 e 4, é possibile fare uso di trait come proposto inizialmente, e poi implementarli sul nuovo tipo.

```
trait OrderedCollectionT {
    fn get_total_items(&self) -> Option<usize>;
}

trait CollectionPageT {
    fn get_next(&self) -> Option<usize>;
}

trait OrderedCollectionPageT: OrderedCollectionT + CollectionPageT {
    fn get_start_index(&self) -> Option<usize>;
}

struct OrderedCollection ( /* ... */ );

struct CollectionPage ( /* ... */ );

struct OrderedCollectionPage ( /* ... */ );

impl OrderedCollectionT for OrderedCollection
    fn get_total_items(&self) -> Option<usize> { /* ... */ }
}

impl CollectionPageT for CollectionPage {
    fn get_next(&self) -> Option<usize> { /* ... */ }
}

impl OrderedCollectionT for OrderedCollectionPage
    fn get_total_items(&self) -> Option<usize> { /* ... */ }
}

impl CollectionPageT for OrderedCollectionPage {
    fn get_next(&self) -> Option<usize> { /* ... */ }
}

impl OrderedCollectionPageT for OrderedCollectionPage {
    fn get_start_index(&self) -> Option<usize> { /* ... */ }
}
```

Codice BU: Esempio semplificato in cui si implementano `trait` di accesso alle proprietà, postfissati con `T`, alle relative `struct`.

Infine, il requisito 7 si é ottenuto creando un trait che astragga l'accesso ai dati, poi rendendo generico il newtype, e richiedendo che il tipo generico implementi quel trait: si é chiamato questo tipo `LDSource`, da «linked data source».

```rust
trait LDSource { /* ... */ }

struct OrderedCollectionPage<Source> (pub Source)
    where Source: LDSource;

impl LDSource for Box<json_ld::Node> { /* ... */ }
```

Codice BV: Esempio in cui si rende generico il tipo `OrderedCollectionPage`.

4.6.4. Specializzazione dei proxy

La base raggiunta é soddisfacente, ma non ancora ideale, in quanto con essa é sempre richiesto di avere la proprietà dell'oggetto «sorgente» per interagirvi, anche se ad esempio si vuole interagire con esso in sola lettura.

Questo può diventare complicato da gestire nella pratica, perché per le regole della proprietà il proprietario é uno e unico [Sezione 3.1.1.]: ovvero, può esserci al massimo un «lettore» alla volta, ed esso deve essere distrutto prima di poterne creare un altro.

Il problema é stato risolto trasformando il singolo newtype in uno nuovo per ogni varietà di variabile esistente in Rust (di proprietà, riferimento mutabile, riferimento immutabile) e separando funzionalità di lettura e scrittura nel trait originariamente definito in due trait diversi, arrivando ai cinque item indicati in [Tabella C].

Item	Nome	Descrizione
`trait`	`ThingReader`	Il trait che denota che da un certo tipo Rust possono essere lette le proprietà di quel tipo JSON-LD. Contiene metodi per **ottenere riferimenti immutabili** alle proprietà.
`struct`	`ThingRef`	Un nuovo newtype. Contiene un **riferimento immutabile** alla sorgente [Sezione 3.1.3.]. Implementa `ThingReader`.
`trait`	`ThingWriter`	Il trait che denota che da un certo tipo Rust possono essere scritte le proprietà di quel tipo JSON-LD. Contiene metodi per **ottenere riferimenti mutabili**, **modificare**, o **creare** le proprietà.
`struct`	`ThingMut`	Un nuovo newtype. Contiene un **riferimento mutabile** alla sorgente [Sezione 3.1.2.]. Implementa `ThingWriter`.
`struct`	`Thing`	Il newtype creato originariamente in [Codice BS]. **Possiede** la sorgente che contiene [Sezione 3.1.1.]. Implementa **sia** `ThingReader` **sia** `ThingWriter`.

Tabella C: I cinque item Rust definiti per rappresentare un tipo JSON-LD, dove il nome del tipo é indicato come `Thing`.

Alle `struct` definite é stato dato il nome di *proxy*, in quanto servono per operare indirettamente su un altro valore; si é scelto allora di definire dei `trait` che le denotassero:

- `LDReader` per tutti i proxy facenti uso di **riferimento immutabile** alla sorgente;
- `LDWriter` per tutti i proxy facenti uso di **riferimento mutabile** alla sorgente;
- `LDOwned` per tutti i proxy con **proprietà** della sorgente.

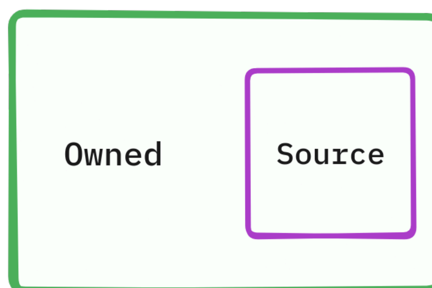

Figura H: Diagramma mostrante la struttura del proxy `Thing` con proprietà della sorgente.

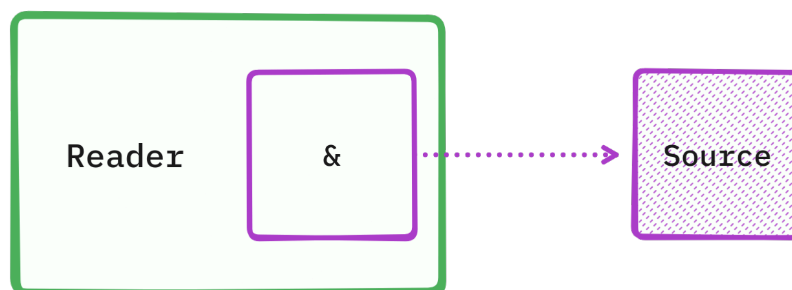

Figura I: Diagramma mostrante la struttura del proxy `ThingRef` con riferimento immutabile alla sorgente.

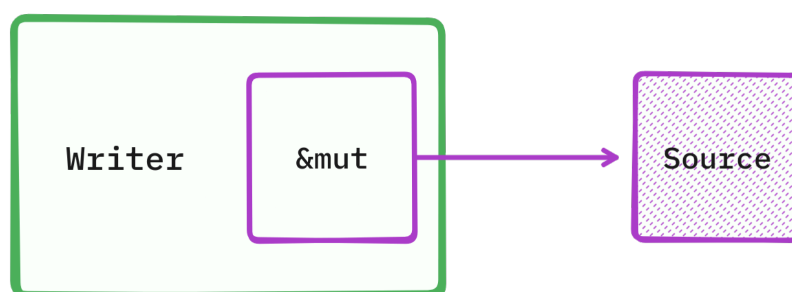

Figura J: Diagramma mostrante la struttura del proxy `ThingMut` con riferimento mutabile alla sorgente.

Come conseguenza del cambiamento, anche i restanti trait di supporto sono cambiati: ad esempio, il `trait LDSource` di [Codice BV] é raddoppiato a sua volta, diventando, `LDSource` e `LDMutSource`, rappresentanti rispettivamente una sorgente dati leggibile e scrivibile.

```
trait LDSource { /* ... */ }

trait LDMutSource: LDSource { /* ... */ }
```

Codice BW: Il `trait Source` di [Codice BV] dopo la triplicazione dei proxy.

4.6.5. Generalizzazione di chiavi e valori

I trait `LDSource` e `LDMutSource` devono permettere l'accesso a proprietà arbitrarie della sorgente, ma essendo trait, non conoscono concretamente che tipi avranno avere le chiavi e i valori delle proprietà.

Senza il requisito 7, si potrebbe direttamente fare uso dei tipi che `json_ld` usa come chiavi e valori, rispettivamente `IriBuf` e `IndexedObject`.

Per soddisfare il requisito 7, é necessaria una generalizzazione.

Per generalizzare le chiavi, si é creato il trait `LDKey`, che garantisce che gli implementatori possano essere copiati (`Clone`), confrontati (`Eq`), utilizzati in una `HashMap` (`Hash`), e costruiti a partire da un IRI (metodo `from_str`) [Sezione 3.10.1.], requisiti che ci si aspetta siano necessari in qualsiasi implementazione di una sorgente dati.

```
/// Denotes that the struct can be used as a linked data key, and
provides the function to convert an IRI into it.
pub trait LDKey: Debug + Clone + Eq + Hash {
    /// Convert an IRI into a linked data key.
    fn from_str(iri: &'static str) -> Self;
}
```

Codice BX: Definizione del tipo `LDKey`.

Per generalizzare i valori, si é creato il trait `LDValue`, da implementare sui valori «di proprietà» [Sezione 3.1.1.], che non offre alcuna garanzia, ma fa uso degli associated types [Sezione 3.1.7.] per collegare l'implementatore ai corrispondenti valori «per riferimento immutabile» (`Ref`) [Sezione 3.1.3.] e «per riferimento mutabile» (`Mut`) [Sezione 3.1.2.].

Dato che nelle implementazioni dei trait i riferimenti devono obbligatoriamente avere un lifetime [Sezione 3.1.4.] esplicito, si é descritto il lifetime del tipo «sorgente» da cui i riferimenti provengono con `'src` («source»).

```
/// Marker for any type representing a writable linked data value.
pub trait LDValue<'src> {
    /// A reference of the same type.
    type Ref: 'src;

    /// A mutable reference of the same type.
    type Mut: 'src;
}
```

Codice BY: Definizione del tratto `LDValue`. Gli associated type `Ref` e `Mut` sono vincolati ad avere un lifetime di almeno `'src`.

```
impl<'src> LDValue<'src> for IndexedObject {
    type Ref = &'src IndexedObject;
    type Mut = &'src mut IndexedObject;
}
```

Codice BZ: Implementazione di `LDValue` per il tipo `IndexedObject` della crate `json_ld`. Come da vincolo, `Ref` e `Mut` hanno un lifetime di `'src`.

```rust
impl<'src> LDValue<'src> for String {
    type Ref = &'src str;
    type Mut = &'src mut str;
}
```

Codice CA: Ipotetica implementazione di `LDValue` per `String`, per una ipotetica source `HashMap<String, String>`. Si evidenzia come i tipi di `Ref` e `Mut` siano diversi (`str`) dal tipo implementatore (`String`)!

Poi, si sono definiti due associated type omonimi in `LDSource`, e li si sono vincolati con il rispettivo trait, aggiungendo il lifetime `'src` anche a `LDSource` e definendo poi una funzione `get_ld_values` per permettere l'accesso ai valori come desiderato.

```rust
pub trait LDSource<'src>
{
    /// The item with which the source can be queried.
    type LDKey: LDKey;

    /// The resulting value of a query.
    type LDValue: LDValue<'src>;

    // ...

    /// Query the source with the given [`Self::LDKey`] for linked data
values, which are returned as a [`Vec`] of references to
[`Self::LDValue`]s.
    fn get_ld_values(&'src self, key: &Self::LDKey) ->
Vec<<Self::LDValue as LDValue<'src>>::Ref>;

    // ...
}
```

Codice CB: Estratto della definizione di `LDSource`, in cui si evidenziano `LDKey` e `LDValue`.

Stessa cosa si é poi fatta con `LDMutSource`, aggiungendo però stavolta i metodi `get_ld_values_mut`, `set_ld_values` e `remove_ld_values` per la modifica.

4.6.6. Interazione con tipi JSON-LD

I trait `LDSource` e `LDMutSource` però non possono limitarsi a permettere interazione con le *proprietà* di una sorgente; devono permettere anche l'elaborazione dei *tipi JSON-LD*.

Si sono allora definiti i trait `LDTypes` e `LDMutTypes`, indicanti che l'implementatore rispettivamente «ha» dei tipi JSON-LD che si possono rispettivamente leggere e scrivere.

Sapendo che tutti i tipi sono IRI [Sezione 3.10.6.], ovvero chiavi, si é definito l'associated type `LDType` descrivente il tipo Rust con cui é rappresentato un tipo JSON-LD, e lo si é vincolato con `LDKey`.

Si sono poi aggiunti metodi per interagire con i tipi:

- su `LDTypes`, `has_ld_type` e `get_ld_types`, per verificare la presenza di un tipo JSON-LD specifico o ottenere tutti i tipi;
- su `LDMutTypes`, `add_ld_type` e `remove_ld_type`, per rispettivamente aggiungere o rimuovere un tipo dalla sorgente.

```
/// Item with infallibly queryable linked data types.
pub trait LDTypes {
    /// The value of a linked data type.
    type LDType: LDKey;

    /// Check if the given `value` is present among the linked data
types of the item.
    fn has_ld_type(&self, value: &Self::LDType) -> bool;

    /// Get an [`HashSet`] of all the linked data types of the item.
    fn get_ld_types(&self) -> HashSet<Self::LDType>;
}
```

Codice CC: Definizione di `LDTypes`.

```
/// Item with infallibly modifiable linked data types.
pub trait LDMutTypes
where
    Self: LDTypes,
{
    /// Add a new linked data type to the item.
    ///
    /// Returns `false` if the type was not added because it already
existed.
    ///
    fn add_ld_type(&mut self, value: Self::LDType) -> bool;

    /// Remove an existing linked data type to the item.
    ///
    /// Returns `false` if the type was not removed because it didn't
exist.
    ///
    fn remove_ld_type(&mut self, value: Self::LDType) -> bool;
}
```

Codice CD: Definizione di `LDMutTypes`.

4.6.7. Tipi JSON-LD associati ai proxy

Si può dire che un proxy possieda i tipi JSON-LD della sorgente che esso contiene, ma non solo: il tipo stesso ha una associazione statica con il tipo (o i tipi) che rappresentano.

Si sono allora estesi i trait `LDTypes` e `LDMutTypes` per includerla, creando i trait `AssociatedLDTypes` e `AssociatedLDMutTypes`.

```rust
/// Associates linked data types to an item.
pub trait AssociatedLDTypes
where
    Self: Sized + LDTypes,
{
    /// The specific linked data types associated to the implementor.
    fn associated_ld_types() -> HashSet<Self::LDType>;

    // ...
}
```

Codice CE: Estratto della definizione di `AssociatedLDTypes`. `associated_ld_types` restituisce un `HashSet` di proprietà nonostante sia un metodo statico per via di problemi inesplorati riscontrati durante la sua realizzazione con un riferimento immutabile statico.

`AssociatedLDTypes` ha un metodo statico, `associated_ld_types`, che restituisce l'insieme dei tipi JSON-LD rappresentati dall'implementatore, e vari altri metodi per determinare se questi tipi sono presenti o no su un'istanza di esso, restituendo il risultato in forme diverse:

- `missing_associated_ld_types`, che restituisce i tipi JSON-LD che mancano;
- `has_associated_ld_types`, che restituisce `true` se sono presenti tutti i tipi JSON-LD, e `false` altrimenti;
- `verify_associated_ld_types`, che restituisce `Ok` se sono presenti tutti i tipi, e un `Err` contenente i tipi mancanti altrimenti;
- `check_associated_ld_types`, che fa la stessa cosa ma prendendo proprietà del tipo, e restituendolo in caso di `Ok`, costituendo un'interfaccia «fluent».

`AssociatedLDMutTypes` inoltre ha:

- `add_associated_ld_types`, che prova ad aggiungere i tipi JSON-LD associati;
- `ensure_associated_ld_types`, che fa la stessa cosa ma con interfaccia «fluent»;
- `add_missing_associated_ld_types`, che prova ad aggiungere solo i tipi JSON-LD associati che `missing_associated_ld_types` riporta manchino.

Visto che `AssociatedLDMutTypes` non contiene metodi che devono essere completati dall'implementatore, lo si é poi *implementato a tappeto* (in inglese *blanket implementation*, un'implementazione automatica di un trait per tutti i tipi soddisfacenti un certo vincolo) per tutti i tipi implementanti sia `LDMutTypes` sia `AssociatedLDTypes`.

```rust
impl<Any> AssociatedLDMutTypes for Any where Any: LDMutTypes +
AssociatedLDTypes {}
```

*Codice CF: L'*implementazione a tappeto *di `AssociatedLDMutTypes`.*

4.6.8. Creazione di funzionalità di conversione

Per il requisito 5, all'accesso alle proprietà di un tipo JSON-LD, si vuole che i proxy restituiscano un tipo Rust significativo per il valore della stessa, come `Vec<String>` o `u64` o `LengthUnit` [Sezione 4.8.1.], e non un tipo generico come `Vec<IndexedObject>` o in

generale un vettore di implementatori di `LDValue` usato dalla sorgente nella sua rappresentazione interna.

É quindi necessario un modo per convertire da rappresentazione interna a tipo desiderato e viceversa, tenendo conto di possibili errori di conversione.

Si presenta nuovamente la triplicazione già vista in [Sezione 4.6.3.] e [Sezione 4.6.5.]:
- per leggere una proprietà JSON-LD, si vogliono ottenere **riferimenti immutabile** ai suoi valori;
- per modificare una proprietà JSON-LD, si vogliono ottenere **riferimenti mutabili** ai suoi valori;
- per impostare una proprietà JSON-LD, si vogliono ricevere **valori di cui si ha ownership** da inserirvi.

In modo simile a come si é fatto con `LDValue`, si é scelto di creare un trait `LDConvert` per effettuare queste conversioni, contenente associated type [Sezione 3.1.7.] per ogni tipo coinvolto.

Ricordando che le proprietà JSON-LD sono considerate aventi sempre un array di valori, come visto in [Codice AA], si considerano essere `Vec` i tipi della rappresentazione interna, e si ricorda che essi, essendo valori di una `LDSource`, implementano `LDValue`; si generalizza quindi il trait `LDConvert` con il tipo di un singolo valore `Value`, vincolandolo a `LDValue`.

```
pub trait LDConvert<'src, Value>
where
    Value: LDValue<'src>
{
    // ...
}
```

Codice CG: Estratto della definizione del trait `LDConvert`, che mostra il tipo generico `Value`.

Partendo dall'ultimo caso, più semplice, dell'impostazione dei valori della proprietà: si é definita la funzione `try_ld_set` (dal nome simile a `TryFrom::try_from`, per le conversioni fallibili), che riceve in input il valore del tipo desiderato, la cui forma posseduta si é definita come tipo associato `Set`, e restituisce in output o il valore convertito con successo a un `Vec` di `Value`, o un errore di conversione, del tipo associato definito `SetErr`.

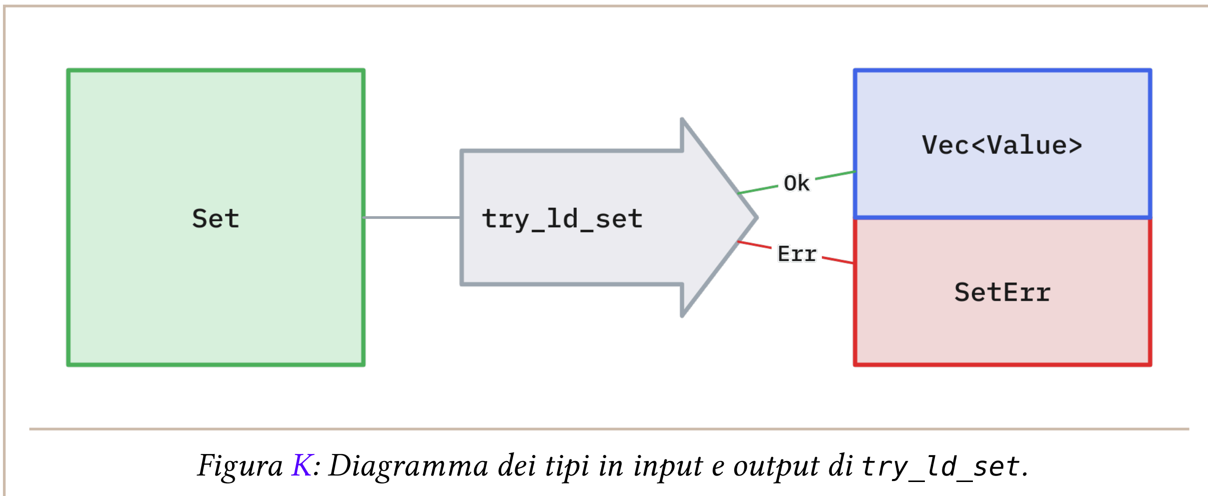

Figura K: Diagramma dei tipi in input e output di `try_ld_set`.

```
pub trait LDConvert<'src, Value> /* ... */ {
    // ...

    /// The type to require as parameter when a conversion from
[`Self::Set`] to [`Vec`] of [`LDValue`] succeeds.
    type Set;

    /// The type to return as [`Err`] variant when a conversion from
[`Self::Set`] to [`Vec`] of [`LDValue`] fails.
    type SetErr: Error;

    /// Convert a [`Self::Set`] into a [`Vec`] of owned values.
    fn try_ld_set(values: Self::Set) -> Result<Vec<Value>,
Self::SetErr>;

    // ...
}
```

Codice CH: Estratto della definizione del trait `LDConvert`, che mostra il metodo `try_ld_set` e i relativi tipi associati.

Passando al primo caso, dell'accesso ai valori in lettura: si é definita la funzione `try_ld_ref`, che riceve in input un `Vec` di riferimenti a valori della sorgente del tipo `Value::Ref`, e restituisce in output o il tipo desiderato `Ref`, o un errore di conversione `RefErr`.

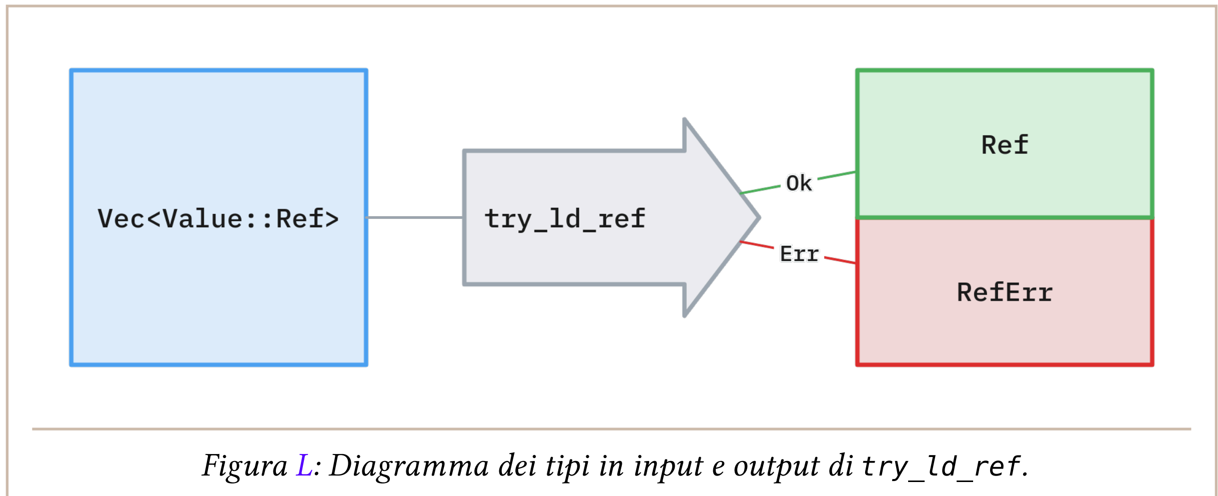

Figura L: Diagramma dei tipi in input e output di `try_ld_ref`.

```
pub trait LDConvert<'src, Value> /* ... */ {
    // ...

    /// The type to return as [`Ok`] variant when a conversion from
[`Vec`] of [`LDValue::Ref`] to [`Self::Ref`] succeeds.
    type Ref;

    /// The type to return as [`Err`] variant when a conversion from
[`Vec`] of [`LDValue::Ref`] to [`Self::Ref`] fails.
    type RefErr: Error;

    /// Convert a [`Vec`] of references to values into [`Self::Ref`].
    fn try_ld_ref(values: Vec<Value::Ref>) -> Result<Self::Ref,
Self::RefErr>;

    // ...
}
```

Codice CI: Estratto della definizione del trait `LDConvert`, che mostra il metodo `try_ld_ref` e i relativi tipi associati.

Infine, nel secondo caso, l'accesso ai valori in scrittura: si é definita la funzione `try_ld_mut`, con un `Vec` di `Value::Mut` in input e in output o il tipo desiderato `Mut`, o l'errore di conversione `MutErr`.

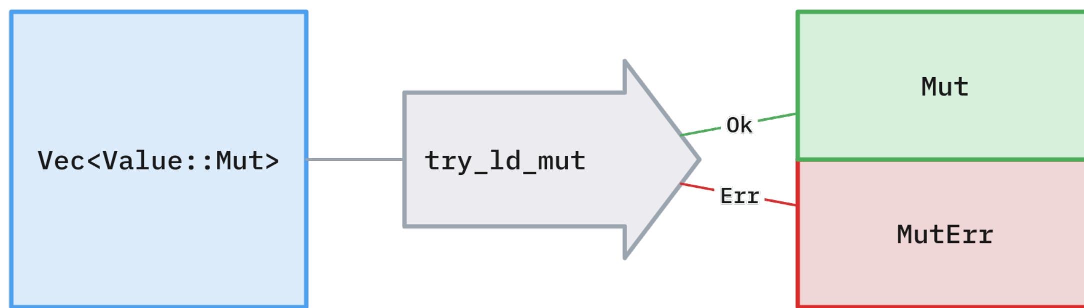

Figura M: Diagramma dei tipi in input e output di `try_ld_mut`.

```
pub trait LDConvert<'src, Value> /* ... */ {
    // ...

    /// The type to return as [`Ok`] variant when a conversion from
[`Vec`] of [`LDValue::Mut`] to [`Self::Mut`] succeeds.
    type Mut;

    /// The type to return as [`Err`] variant when a conversion from
[`Vec`] of [`LDValue::Mut`] to [`Self::Ref`] fails.
    type MutErr: Error;

    /// Convert a [`Vec`] of mutable references to values into
[`Self::Mut`].
    fn try_ld_mut(values: Vec<Value::Mut>) -> Result<Self::Mut,
Self::MutErr>;

    // ...
}
```

Codice CJ: Estratto della definizione del trait `LDConvert`, che mostra il metodo `try_ld_ref` e i relativi tipi associati.

```
/// Converts between a [`Vec`] of [`LDValue`]s and the types
[`Self::Ref`], [`Self::Mut`], and [`Self::Set`].
///
/// Allows to use the implementing item as the type of a `vocab` field.
///
pub trait LDConvert<'src, Value>
where
    Value: LDValue<'src>,
{
    /// The type to return as [`Ok`] variant when a conversion from
[`Vec`] of [`LDValue::Ref`] to [`Self::Ref`] succeeds.
    type Ref;

    /// The type to return as [`Ok`] variant when a conversion from
[`Vec`] of [`LDValue::Mut`] to [`Self::Mut`] succeeds.
    type Mut;

    /// The type to require as parameter when a conversion from
[`Self::Set`] to [`Vec`] of [`LDValue`] succeeds.
    type Set;

    /// The type to return as [`Err`] variant when a conversion from
[`Vec`] of [`LDValue::Ref`] to [`Self::Ref`] fails.
    type RefErr: Error;

    /// The type to return as [`Err`] variant when a conversion from
[`Vec`] of [`LDValue::Mut`] to [`Self::Ref`] fails.
    type MutErr: Error;

    /// The type to return as [`Err`] variant when a conversion from
[`Self::Set`] to [`Vec`] of [`LDValue`] fails.
    type SetErr: Error;

    /// Convert a [`Vec`] of references to values into [`Self::Ref`].
    fn try_ld_ref(values: Vec<Value::Ref>) -> Result<Self::Ref,
Self::RefErr>;

    /// Convert a [`Vec`] of mutable references to values into
[`Self::Mut`].
    fn try_ld_mut(values: Vec<Value::Mut>) -> Result<Self::Mut,
Self::MutErr>;

    /// Convert a [`Self::Set`] into a [`Vec`] of owned values.
    fn try_ld_set(values: Self::Set) -> Result<Vec<Value>,
Self::SetErr>;
}
```

Codice CK: Definizione completa di `LDConvert`.

Successivamente, si sono aggiunti nuovi metodi a `LDSource` e `LDMutSource`, in modo da rendere possibile ottenere valori ed applicare una relativa conversione con una sola chiamata: `get_ld_item`, `get_ld_item_mut` e `set_ld_item`, corrispondenti rispettivamente a `get_ld_values`, `get_ld_values_mut` e `set_ld_values`.

```
pub trait LDSource<'src>
where
    Self: LDTypes,
{
    // ...

    /// Query the source with the given [`Self::LDKey`] for linked data
values, which are returned as `Item::Ref`, thanks to a conversion with
[`LDConvert::try_ld_rev`].
    fn get_ld_item<Item>(&'src self, key: &Self::LDKey) ->
Result<Item::Ref, Item::RefErr>
    where
        Item: LDConvert<'src, Self::LDValue>,
    {
        Item::try_ld_ref(self.get_ld_values(key))
    }
}
```

Codice CL: Estratto della definizione del metodo `LDSource::get_ld_item`.

```
pub trait LDSource<'src>
where
    Self: LDSource<'src> + LDMutTypes,
{
    // ...

    /// Query the source with the given [`Self::LDKey`] for linked data
values, which are returned as `Item::Mut`, thanks to a conversion with
[`LDConvert::try_ld_mut`].
    fn get_ld_item_mut<Item>(&'src mut self, key: &Self::LDKey) ->
Result<Item::Mut, Item::MutErr>
    where
        Item: LDConvert<'src, Self::LDValue>,
    {
        Item::try_ld_mut(self.get_ld_values_mut(key))
    }

    /// ...

    /// Set and overwrite the linked data values of the given
[`Self::LDKey`] to the given `Item::Set`, thanks to a conversion with
[`LDConvert::try_ld_set`].
    fn set_ld_item<Item>(&mut self, key: Self::LDKey, item: Item::Set) -
> Result<(), Item::SetErr>
    where
        Item: LDConvert<'src, Self::LDValue>,
    {
        let values = Item::try_ld_set(item)?;
        self.set_ld_values(key, values);
        Ok(())
    }
}
```

Codice CM: Estratto della definizione dei metodi `LDMutSource::get_ld_item_mut` e `LDMutSource::set_ld_item`.

4.6.9. Appiattimento dei valori con vettori innestati

Nella grande maggioranza dei casi, visto che le funzioni di `LDConvert` trattano `Vec<Value>`, avranno `Vec<...>` dal lato opposto della conversione: dover sempre specificare `Vec` potrebbe quindi risultare tedioso.

Si é allora definito un altro trait, `LDConvertSingle`, dedicato alla conversione di una singola `Value` in un singolo tipo desiderato, in modo che in fase di realizzazione pratica [Sezione 4.6.10.] si potesse implementare a tappeto `LDConvert` per chiunque implementasse `LDConvertSingle`.

```
pub trait LDConvertSingle<'src, Value>
where
    Value: LDValue<'src>,
{
    // ...

    /// Convert a reference to a value into [`Self::Ref`].
    fn try_ld_ref(value: Value::Ref) -> Result<Self::Ref, Self::RefErr>;

    /// Convert a mutable reference to a value into [`Self::Mut`].
    fn try_ld_mut(value: Value::Mut) -> Result<Self::Mut, Self::MutErr>;

    /// Convert a [`Self::Set`] into a owned value.
    fn try_ld_set(value: Self::Set) -> Result<Value, Self::SetErr>;
}
```

Codice CN: Estratto della definizione di `LDConvertSingle`, che evidenzia ccome sono cambiati i metodi `try_ld_ref`, `try_ld_mut` e `try_ld_set` da `LDConvert`.

4.6.10. Realizzazione pratica dell'astrazione con la crate `json_ld`

Avendo definito i trait descriventi l'astrazione da implementare, si é finalmente potuti passare all'implementazione con una sorgente dati concreta.

Si é creato un nuovo modulo, `impls`, che contenesse tutte le implementazioni pratiche realizzate per diverse sorgenti di dati, e al suo interno si é creato un ulteriore modulo, `json_ld`, che contenesse l'implementazione realizzata con l'omonima crate.

```
#[cfg(feature = "json-ld")]
pub mod json_ld;
```

Codice CO: Definizione del sottomodulo `json_ld`. L'attributo `#[cfg(feature = "json-ld")]` lo include nella build solamente se la feature con quel nome é abilitata, rendendo così la crate `json_ld` una dipendenza opzionale.

Nel modulo si sono realizzati tanti altri sottomoduli, uno per ogni tipo per cui si sono implementati dei trait.

4.6.11. Conversione identità

Il primo sottomodulo realizzato é stato `jld_indexedobject`, che tratta il tipo «valore» utilizzato dalla crate `json_ld`, `Indexed<Object<IriBuf, BlankIdBuf>>`; in esso:

- é stato creato un alias `IndexedObject` per il tipo «valore»;
- si é implementato `LDValue` per esso come visto in [Codice BZ];
- si é implementato `LDConvert` per un `Vec` di esso, in modo tale che `Vec<IndexedObject>` potesse essere usato come tipo parametrico delle funzioni `_item` [Sezione 4.6.8.], in modo da poterlo usare direttamente senza elaborazioni.

```
impl<'src> LDConvert<'src, IndexedObject> for Vec<IndexedObject> {
    type Ref = Vec<&'src IndexedObject>;
    type Mut = Vec<&'src mut IndexedObject>;
    type Set = Vec<IndexedObject>;
    type RefErr = Infallible;
    type MutErr = Infallible;
    type SetErr = Infallible;

    fn try_ld_ref(values: Vec<&'src IndexedObject>) -> Result<Self::Ref,
Self::RefErr> {
        Ok(values)
    }

    fn try_ld_mut(values: Vec<&'src mut IndexedObject>) ->
Result<Self::Mut, Self::MutErr> {
        Ok(values)
    }

    fn try_ld_set(values: Self::Set) -> Result<Vec<IndexedObject>,
Self::SetErr> {
        Ok(values)
    }
}
```

Codice CP: L'implementazione di `LDConvert` per `Vec<IndexedObject>`. I metodi non effettuano nessuna elaborazione, e si limitano a restituire l'input all'interno di un risultato `Ok`. Si può notare come il tipo di errore utilizzato sia `Infallible`, un tipo speciale di cui il compilatore sa che non possono esistere istanze, e quindi ottimizza via.

4.6.12. Conversione di appiattimento

Il secondo sottomodulo realizzato é stato `valuenode`, contenente i tipi artificiali `ValueNode`, `ValueNodeRef`, e `ValueNodeMut`, costruiti per fare da ponte tra JSON-LD, che ha la possibilità di avere liste innestate come valore di una proprietà, e il trait `LDConvertSingle`, che per implementare a tappeto `LDConvert` ha bisogno di un singolo vettore di scalari.

```
pub enum Object {
  Value(json_ld::Value),
  Node(Box<json_ld::Node>),
  List(json_ld::List),
}
```

```
pub enum ValueNode {
    Value(json_ld::Value),
    Node(Box<json_ld::Node>),
}
```

Codice CQ: Estratto della definizione di `ValueNode`, confrontato con un estratto della definizione di `json_ld::Object`. Si osserva che `ValueNode` non prevede più la variante `List`.

Pensando a come potrebbero essere utilizzati i metodi `_item`, ci si é ricordati che alcune delle proprietà dei tipi di ActivityStreams sono definite come «funzionali» [111], ovvero

che esse possono avere un solo, singolo, valore; si é pensato allora essere opportuno fare in modo che i metodi _item potessero restituire sia Vec<...> per i casi a più valori, ma anche Option<...> per i casi con un valore singolo.

Si é scelto di realizzare una **doppia** implementazione a tappeto di LDConvert: ogni tipo T per cui é implementato LDConvertSingle implementerà anche LDConvert per Vec<T> e per Option<T>, la prima chiamando il relativo metodo di LDConvertSingle per ogni valore della proprietà, e la seconda effettuandolo solo per il primo.

É allora necessario decidere come trattare i casi in cui un valore (Object) in realtà é una lista di valori (Object::List): si é scelto di trattarli diversamente nei casi di Vec<T> e Option<T>.

Nel caso di Vec<T>, si é scelto di fare uso della ricorsione per appiattire tutte le liste innestate a una sola, facendo diventare ad esempio [[A], [[B], C], D] il Vec [A, B, C, D], chiamando questa funzione vec_from_flattening e implementandola staticamente per i tre tipi ValueNode.

Nel caso di Option<T>, si é scelto di fare uso della ricorsione per effettuare una depth-first search che restisse il valore del primo elemento, facendo diventare ad esempio [[A], [[B], C], D] il valore A, chiamando invece questa funzione from_flattening_first e sempre implementandola staticamente per i tre tipi ValueNode.

```
/// Apply [`LDConvertSingle`] to each item of the input, then collect
the inputs into a [`Vec`].
impl<'src, Convert> LDConvert<'src, IndexedObject> for Vec<Convert>
where
    Convert: LDConvertSingle<'src, ValueNode>,
    ValueNode: LDValue<'src, Ref = ValueNodeRef<'src>, Mut =
ValueNodeMut<'src>>,
{
    type Ref = Vec<Convert::Ref>;
    type Mut = Vec<Convert::Mut>;
    type Set = Vec<Convert::Set>;
    type RefErr = Convert::RefErr;
    type MutErr = Convert::MutErr;
    type SetErr = Convert::SetErr;

    fn try_ld_ref(values: Vec<&'src IndexedObject>) -> Result<Self::Ref,
Self::RefErr> {
        ValueNodeRef::vec_from_flattening(values)
      .into_iter()
      .map::<Result<Convert::Ref, Self::RefErr>, fn(ValueNodeRef<'src>)
-> Result<Convert::Ref, Self::RefErr>>(Convert::try_ld_ref)
      .collect()
    }

    fn try_ld_mut(values: Vec<&'src mut IndexedObject>) ->
Result<Self::Mut, Self::MutErr> {
        ValueNodeMut::vec_from_flattening(values)
      .into_iter()
      .map::<Result<Convert::Mut, Self::MutErr>, fn(ValueNodeMut<'src>)
-> Result<Convert::Mut, Self::MutErr>>(Convert::try_ld_mut)
      .collect()
    }

    fn try_ld_set(values: Self::Set) -> Result<Vec<IndexedObject>,
Self::SetErr> {
        values
      .into_iter()
      .map::<Result<ValueNode, Self::SetErr>, fn(Convert::Set) ->
Result<ValueNode, Self::SetErr>>(Convert::try_ld_set)
      .map(|value| value.map(Into::<IndexedObject>::into))
      .collect()
    }
}
```

Codice CR: L'implementazione di `LDConvert` per `Vec<T>`, dove `T` é qualsiasi tipo implementante `LDConvertSingle`.

```
/// Apply [`LDConvertSingle`] to the first item of the input, then
create an [`Option`] out of it.
impl<'src, Convert> LDConvert<'src, IndexedObject> for Option<Convert>
where
    Convert: LDConvertSingle<'src, ValueNode>,
    ValueNode: LDValue<'src, Ref = ValueNodeRef<'src>, Mut =
ValueNodeMut<'src>>,
{
    type Ref = Option<Convert::Ref>;
    type Mut = Option<Convert::Mut>;
    type Set = Option<Convert::Set>;
    type RefErr = Convert::RefErr;
    type MutErr = Convert::MutErr;
    type SetErr = Convert::SetErr;

    fn try_ld_ref(values: Vec<&'src IndexedObject>) -> Result<Self::Ref,
Self::RefErr> {
        if cfg!(debug_assertions) && values.len() > 1 {
            log::warn!(
                "Attempting to convert more than one Value into an
Option; this is a lossy operation, and only the first will be returned
as `Some`. Consider converting the field to a Vec instead, since
LDConvert is also implemented for it. (This check will not be performed
in production builds.)"
            );
        }

        match
ValueNodeRef::from_flattening_first(values.into_iter().next()) {
            None => Ok(None),
            Some(vn) => Ok(Some(Convert::try_ld_ref(vn)?)),
        }
    }

    // ...
}
```

Codice CS: Estratto dell'implementazione di `LDConvert` per `Option<T>`, dove T é qualsiasi tipo implementante `LDConvertSingle`. Si sono aggiunti dei warning per le build di debug nel caso in cui il tipo letto come `Option<T>` contenga più di un valore.

4.6.13. Conversioni di uso comune

Successivamente, si é implementato `LDConvertSingle` su numerosi tipi di uso comune, appoggiandosi per alcuni sulle implementazioni di quelli precedenti, andando a formare la gerarchia delineata in [Figura N].

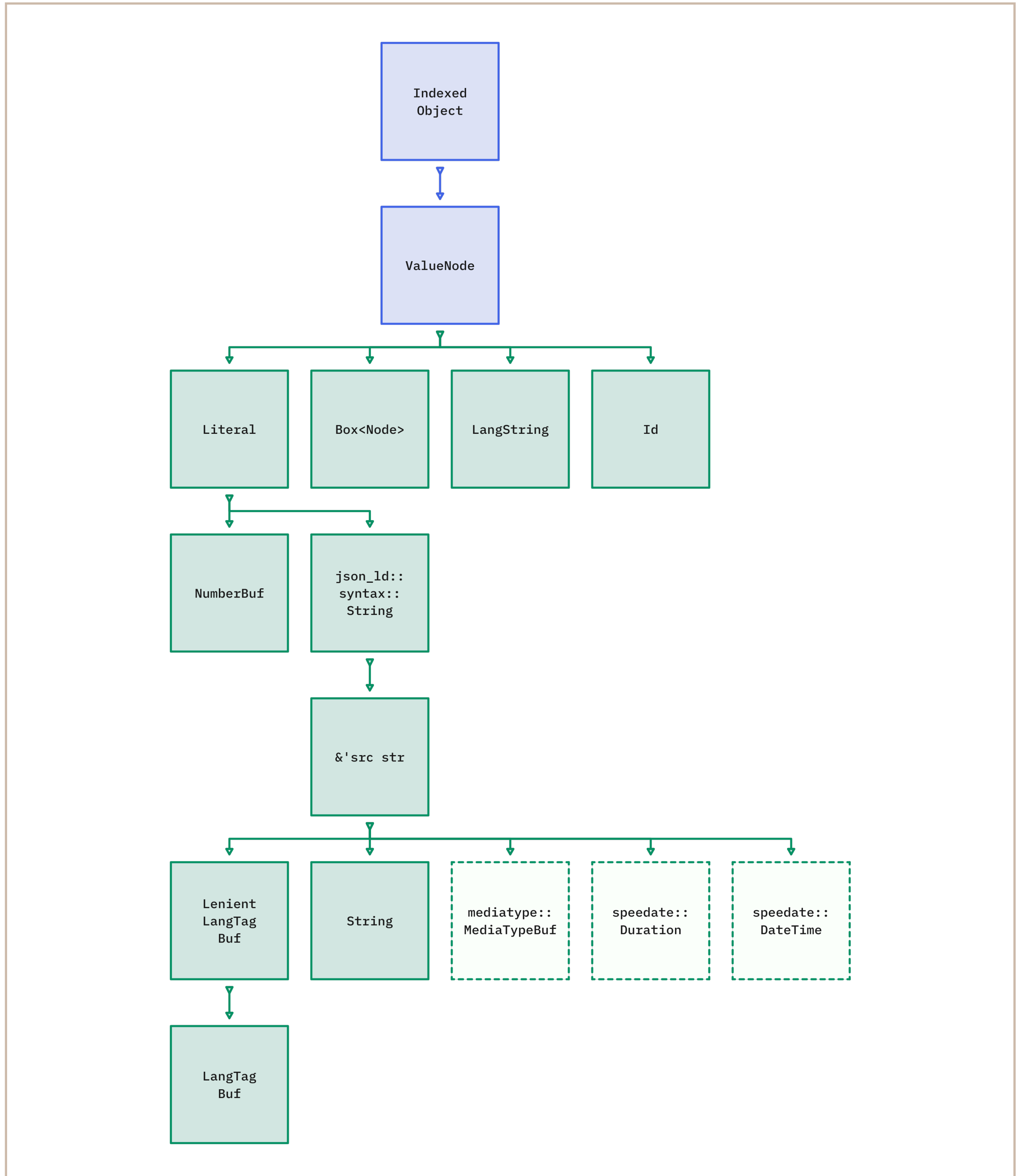

Figura N: Gerarchia finale delle conversioni implementate in `acrate_jsonld::impls::json_ld`. I blocchi blu implementano `LDConvert` direttamente, mentre quelli verdi attraverso `LDConvertSingle`. I blocchi tratteggiati sono inclusi solo se la rispettiva dipendenza é attivata attraverso le feature della crate.

4.7. Realizzazione di macro per implementare le astrazioni JSON-LD

Si é determinato che scrivere il codice di definizione item e delle loro implementazioni come descritto in [Tabella C] per qualsiasi tipo JSON-LD non-triviale manualmente avrebbe richiesto una quantità eccessiva di boilerplate, e sarebbe stato esageratamente tedioso, se non impossibile da realizzare; ricordando che si stava prendendo in considerazione ActivityStreams [Sezione 3.11.] come formato dati, avente una cinquantina di tipi JSON-LD diversi, si é capito che si sarebbe dovuto automatizzare qualcosa.

Essendo questo tipo di problema abbastanza comune in Rust, il linguaggio mette a disposizione la possibilità di scrivere macro [Sezione 3.1.12.] per eliminare la necessità di boilerplate; si é allora provato a scrivere una macro che generasse automaticamente il boilerplate necessario, ricevendo in input solo informazioni riguardanti il tipo JSON-LD, come nome, proprietà, e tipo Rust desiderato per la rappresentazione di esse.

4.7.1. Iterazione iniziale attraverso funzione macro dichiarativa

L'idea iniziale é stata di realizzare una funzione macro dichiarativa [Sezione 3.1.13.] che generasse le implementazioni generate.

Si é riusciti parzialmente nello scopo: gli item venivano generati correttamente, ma, essendo le macro dichiarative stateless, non potevano implementare propriamente il requisito 2, perché ogni chiamata della macro non era a conoscenza delle chiamate precedenti.

Inoltre, le macro dichiarative non sono in grado di generare nuovi identificatori loro stesse, quindi era necessario specificare manualmente i nomi di tutti gli item generati.

```
vocab! {
  "https://www.w3.org/ns/activitystreams#Activity",
  ref ActivityRef,
  mut ActivityMut,
  children [
    {
      "https://www.w3.org/ns/activitystreams#object",
      get_objects,
      into_objects_mut,
      set_objects,
    }
  ],
}
```

Codice CT: Chiamata alla macro `vocab!` in una vecchia versione del codice in cui molte delle astrazioni descritte non esistevano ancora. Il tipo `Activity` ha solo i metodi specifici alla propria proprietà `object`, e non quelli delle proprietà dei tipi da cui eredita, come `subject`.

4.7.2. Iterazione finale attraverso attributo macro procedurale

Si é allora scelto di fare uso di macro procedurali [Sezione 3.1.14.], che eseguendo codice Rust al momento dell'espansione sono in grado di preservare informazioni tra una chia-

mata e l'altra facendo uso di variabili `static`, e quindi di implementare ereditarietà delle proprietà come desiderato.

Si é scelto di usare la forma attributo macro procedurale nello specifico in modo tale che si potesse utilizzare per la descrizione dei tipi JSON-LD la sintassi di definizione `struct` con cui un programmatore Rust dovrebbe già essere familiare, con la sola aggiunta di alcuni attributi.

```
#[vocab]
#[inherits(Thing)]
#[iri = "http://schema.org/Person"]
pub struct Person {
    #[iri = "http://schema.org/name"]
    names: Vec<&'src str>,
}
```

Codice CU: Rappresentazione di esempio del tipo JSON-LD `Person` con solo la proprietà `name` inclusa. É utilizzato per effettuare testing di `acrate_jsonld_macros`.

Si é strutturata la generazione del codice in due fasi, `base` e `inheritance`, la prima responsabile per la generazione degli item e dell'implementazione delle proprietà del tipo stesso, mentre la seconda responsabile per l'implementazione delle proprietà ereditate.

4.7.3. Elaborazione delle proprietà del tipo stesso

Per generare il codice delle proprietà proprie, si é realizzata una `struct` chiamata `BaseBlocksBuilder` che attraverso il tratto `From` potesse essere convertita nel `TokenStream` del codice generato.

Essa stessa contiene cinque `Vec` di `TokenStream` che rappresentano i contenuti dei blocchi di codice degli item generati.

```
/// Struct allowing to incrementally build the definition and
implementation blocks for the generated entities.
#[derive(Clone)]
pub(crate) struct BaseBlocksBuilder<'cfg> {
    /// The [`VocabConfig`] that this struct is building blocks for.
    vocab_config: &'cfg VocabConfig,

    /// Contents of the `trait BaseReader {...}` block.
    reader_trait: Vec<TokenStream2>,

    /// Contents of the `impl BaseReader for BaseRef {...}` and `impl
BaseReader for Base {...}` blocks.
    reader_impl: Vec<TokenStream2>,

    /// Contents of the `trait BaseWriter {...}` block.
    writer_trait: Vec<TokenStream2>,

    /// Contents of the `impl BaseWriter for BaseMut {...}` and `impl
BaseWriter for Base {...}` blocks.
    writer_impl: Vec<TokenStream2>,

    /// Contents of the `impl Base {...}` block.
    own_impl: Vec<TokenStream2>,
}
```

Codice CV: Definizione di `BaseBlocksBuilder`. Il tipo `proc_macro2::TokenStream` é chiamato `TokenStream2` per distinguerlo dal tipo della standard library `proc_macro::TokenStream`, di cui non si fa utilizzo.

I `Vec` di `TokenStream` di `BaseBlocksBuilder` iniziano vuoti; per ciascuna proprietà del tipo JSON-LD da implementare, si chiama il metodo `BaseBlocksBuilder::add_field`, passandogli una struttura descrivente il campo della `struct` rappresentante la proprietà, ed esso aggiunge ai `Vec` i blocchi di codice necessari.

```
/// Consume a [`Data`] to create a [`FieldConfig`] for each of its fields.
pub(crate) fn vec_from_data(value: Data) -> syn::Result<Vec<Self>> {
    let fields = match value {
        Data::Enum(d) => {
            Err(e!(d.enum_token.span() => "vocab-tagged entities must be structs."))?
        }
        Data::Union(d) => {
            Err(e!(d.union_token.span() => "vocab-tagged entities must be structs."))?
        }
        Data::Struct(d) => match d.fields {
            Fields::Named(FieldsNamed { named, .. }) => named,
            Fields::Unit => {
                Err(e!(d.struct_token.span() => "vocab-tagged structs must use named fields."))?
            }
            Fields::Unnamed(f) => {
                Err(e!(f.span() => "vocab-tagged structs must use named fields."))?
            }
        },
    };

    fields.into_iter().map(Self::from_field).collect()
}
```

Codice CW: Definizione della funzione `vec_from_data`, che riceve in input i token della `struct` dichiarata, ed emette in output un `Vec` di `FieldConfig` che `BaseBlocksBuilder::add_field` é in grado di elaborare. Questo pezzo di codice nello specifico verifica che l'item a cui é stato aggiunto l'attributo sia una `struct` con campi dal nome esplicito, e non un enum, una union, una `struct` vuota, o una `struct` con campi posizionali, delegando poi il resto dell'elaborazione a `FieldConfig::from_field`.

```
pub(crate) fn add_field(&mut self, field_config: &FieldConfig)
```

Codice CX: La signature di `add_field`.

```
self.reader_impl.push(
    quote_spanned! { field_config.span =>
        fn #ident_fn_get_ref(&'src self) -> Result<
            <#ty as acrate_jsonld::traits::values::LDConvert<'src,
Source::LDValue>>::Ref,
            <#ty as acrate_jsonld::traits::values::LDConvert<'src,
Source::LDValue>>::RefErr
        >
            where
                #ty: acrate_jsonld::traits::values::LDConvert<'src,
Source::LDValue>
        {
            let id = #ident_struct_own::<'src, Source>::#ident_fn_iri();

            self.0.get_ld_item::<#ty>(&id)
        }
    }
);
```

Codice CY: Estratto di codice della funzione `add_field`, che mostra l'aggiunta di un blocco di codice al `Vec reader_impl`. Il codice é stato ri-indentato per renderlo più facilmente leggibile. Gli identificatori prefissati con `#` sono sostituzioni: ad esempio, `#ty` viene sostituito nel codice generato con il tipo del campo a cui il metodo corrisponde. Ad esempio, in [Codice CU], implementando il campo `names`, `#ty` verrebbe sostituito con `Vec<&'src str>`, inserendo così una chiamata a `get_ld_item` a cui é imposto di restituire `Vec<&'src str>`.

In fase di conversione a `TokenStream`, i `Vec` vengono concatenati, poi vengono usati per generare i blocchi di codice boilerplate, effettuando le appropriate sostituzioni al fine che essi abbiano nomi e tipi stabiliti.

```
#vis trait #ident_trait_reader<'src, Source> where
    Source: acrate_jsonld::traits::sources::LDSource<'src> + 'src,
{
    #( #reader_trait )*
}
```

Codice CZ: Breve estratto del template utilizzato per convertire `BaseBlocksBuilder` in un unico `TokenStream`. `#( #reader_trait )*` effettua il concatenamento del `Vec reader_trait`, poi lo usa conme contenuto del blocco di definizione del `trait ThingReader`, dove `Thing` é il nome del tipo JSON-LD.

4.7.4. Considerazioni per l'implementazione dell'ereditarietà

Per implementare le proprietà ereditate si é seguito un approccio simile, ma facendo alcune considerazioni, da cui derivano le limitazioni di `#[vocab]`.

Una `struct` taggata con `#[vocab]`:

1. non é e non può essere a conoscenza delle altre `struct` con lo stesso tag;
2. non può condividere attraverso una variabile `static` dei suoi token con altre chiamate in quanto i token non implementano `Sync` e `Send` e quindi non sono thread-safe;
3. può convertire i suoi token a `String`, che sono thread-safe;
4. può condividere una variabile `static` contenente `String` con altre `struct` taggate;
5. può ri-creare dei token a partire da `String`.

La procedura di trasferimento é lossy: nell'effettuarla, si perdono informazioni sullo `Span`, ovvero la posizione del codice sorgente dei token della `struct` originale, necessarie per indicare dove si trova la causa di un errore; come risultato, qualsiasi errore di compilazione risultante dal codice generato da `inheritance` viene indicato nel primo carattere di `#[vocab]`.

Inoltre, c'é un problema di scope: se un tipo esterno, come ad esempio `MediaTypeBuf` é in scope nella posizione di una definizione `#[vocab]` grazie a uno statement `use mediatype::MediaTypeBuf;`, non é detto che esso lo sia anche nella posizione di una seconda definizione `#[vocab]` che eredita dalla prima, se essa é in un altro modulo, e visto che al momento dell'espansione delle macro i percorsi dei token non sono ancora stati risolti, non c'é nulla che si possa fare per garantire che il tipo sia portato in scope anche nella seconda definizione.

Per evitare il problema di scope, si é specificato nella documentazione che tutti i tipi nelle `struct` taggate con `#[vocab]` devono **fare uso di un percorso globale**, oppure essere in scope in tutte le definizioni di `struct` `#[vocab]` che condividono relazioni di ereditarietà.

4.7.5. Implementazione delle proprietà ereditate

Per prima cosa, si é aggiunta una funzione che, in seguito all'implementazione delle proprietà proprie in `base`, salvasse nella variabile `static` `INHERIT_STORAGE` le informazioni necessarie all'implementazione delle stesse in un altro tipo.

```rust
/// Holder for the global information used to generate implementations
of inherited vocab traits.
pub(crate) static INHERIT_STORAGE: LazyLock<RwLock<InheritStorage>> =
    LazyLock::new(|| RwLock::new(InheritStorage::default()));
```

Codice DA: Definizione della variabile statica `INHERIT_STORAGE`, il cui contenuto é condiviso tra tutte le chiamate alla macro `vocab`.

Si sono poi considerati i due casi in cui ci si può trovare quando il `B` eredita le proprietà dal `A`, non essendo a conoscenza dell'ordine di definizione delle `struct` `#[vocab]` e non potendolo modificare:

- `A` é già stato definito, quindi le informazioni per implementare le proprietà di `A` in `B` sono già disponibili in `INHERIT_STORAGE`;
- `A` non é ancora stato definito, quindi le informazioni relative ad esso sono sconosciute, e bisogna aspettare che venga definito per implementarne le proprietà in `B`, lasciando una nota a riguardo in `INHERIT_STORAGE`.

La `struct InheritStorage` é stata allora costruita con i due seguenti campi:
- `known`, che contiene le informazioni dei tipi già implementati per poterne implementare le proprietà su altri tipi;
- `unknown`, che contiene per ogni tipo di cui non si conoscono le informazioni, i nomi dei tipi su cui dovranno essere implementate le relative proprietà.

```
/// Type of the contents of [`Storage`].
#[derive(Debug, Clone, Default)]
pub(crate) struct InheritStorage {
    /// Mapping from a **known** vocab's stringified [`Ident`] to its
[`KnownVocabInfo`].
    pub known: HashMap<String, KnownVocabInfo>,

    /// Mapping from an **unknown** vocab's stringified [`Ident`] to the
stringifed [`Ident`]s of vocabs inheriting from it.
    pub unknown: HashMap<String, HashSet<String>>,
}
```

Codice DB: Definizione della struct `InheritStorage`, condivisa tra `struct` taggate con `#[vocab]`.

Ricordando che, in presenza di più tipi, come `C` che eredita da `B` che eredita da `A`, implementare le proprietà di `B` per `C` significa anche dover implementare le proprietà di `A` per `C`, si é scelto di fare uso di un algoritmo ricorsivo.

Esso, in ordine:
1. implementa ricorsivamente le proprietà dei tipi da cui il tipo attuale eredita, usando lo storage per rimandare al futuro le implementazioni attualmente impossibili;
2. accede allo storage per determinare quali implementazioni erano precedentemente impossibili senza le informazioni del tipo attuale, implementandole ora ricorsivamente.

Come risultato, si ha la garanzia che le implementazioni siano presenti in qualche parte nel codice, e che quindi siano utilizzabili, ma non si sa dove esse sono state dichiarate: potrebbero essere sotto il tipo che eredita, oppure sotto il tipo ereditato.

```
fn generate_recursion(storage: &mut InheritStorage, current: &str,
inherits: &[String]) -> syn::Result<TokenStream2> {
  let mut streams2 = Vec::<TokenStream2>::new();
  let fields = storage.known.get(current).expect("Expected fields for
the current identifier to be registered").fields.clone();
  // for each item we want to inherit from
  for inherited in inherits {
    // check if the ident is known or unknown
    match storage.known.get_mut(inherited) {
      // ident is unknown
      None => {
        // enqueue it for future implementation
        storage.register_unknown(current.to_string(),
inherited.clone());
      }
      // ident is known
      Some(info) => {
        // if the ident hasn't been processed yet
        if info.implemented_by.insert(current.to_string()) {
          // make sure to not make a double borrow from storage
          let inherited_inherits = info.inherits.to_owned();
          // implement it
          let stream2_self = generate(current, inherited, &info.fields);
          streams2.push(stream2_self);
          // implement all its inherits
          let stream2_sub = generate_recursion(storage, current,
&inherited_inherits)?;
          streams2.push(stream2_sub);
        }
      }
    }
  }
  // check if somebody else is in queue for implementation
  let inheritors = storage.unknown.remove(current).unwrap_or_default();
  for inheritor in inheritors {
    // implement it
    let stream2_self = generate(&inheritor, current, &fields);
    streams2.push(stream2_self);
    // run the process again for the inheritor
    let tokens = generate_recursion(storage, &inheritor, inherits)?;
    streams2.push(tokens);
  }
  Ok(streams2.into_iter().collect())
}
```

Codice DC: La definizione completa della funzione generate_recursion.

4.8. Generazione delle implementazioni dei tipi di ActivityStreams

Completata la realizzazione di `acrate_jsonld` e `acrate_jsonld_macros`, si é anche realizzato il modulo per ActivityStreams [Sezione 3.11.] ipotizzato.

4.8.1. Tipi specializzati

Per prima cosa, si é creata una rappresentazione per i tipi primitivi utilizzati in ActivityStreams ma ancora mancanti perché specifici ad esso: si tratta di un tipo solo, `LengthUnit`, utilizzato per specificare l'unità di misura dei valori di lunghezza del tipo in cui si trova.

```
/// A length unit.
#[derive(Debug, Clone, PartialEq, Eq, Default)]
pub enum LengthUnit {
    /// Centimeters, `"cm"`.
    Centimeters,

    /// Feet, `"feet"`.
    Feet,

    /// Inches, `"inches"`.
    Inches,

    /// Kilometers, `"km"`.
    Kilometers,

    /// Meters, `"m"` or unspecified.
    #[default]
    Meters,

    /// Miles, `"miles"`.
    Miles,

    /// Any other unit, typically in form of an URI.
    Other(String),
}
```

Codice DD: Definizione del tipo `LengthUnit`, usato per la proprietà `unit` del tipo ActivityStreams `Place`.

```
impl From<String> for LengthUnit {
    fn from(value: String) -> Self {
        match value.as_str() {
            "cm"     => Self::Centimeters,
            "feet"   => Self::Feet,
            "inches" => Self::Inches,
            "km"     => Self::Kilometers,
            "m"      => Self::Meters,
            "miles"  => Self::Miles,
            _        => Self::Other(value),
        }
    }
}
```

Codice DE: Implementazione di `From<String>` per il tipo `LengthUnit`.

```
impl<'src> LDConvertSingle<'src, ValueNode> for LengthUnit
where /* ... */,
{
    /* ... */

    fn try_ld_ref(value: ValueNodeRef<'src>) -> Result<Self::Ref,
Self::RefErr> {
        <String as LDConvertSingle<'src, ValueNode>>::try_ld_ref(value)
    }

    fn try_ld_mut(value: ValueNodeMut<'src>) -> Result<Self::Mut,
Self::MutErr> {
        <String as LDConvertSingle<'src, ValueNode>>::try_ld_mut(value)
    }

    fn try_ld_set(value: Self::Set) -> Result<ValueNode, Self::SetErr> {
        <String as LDConvertSingle<'src, ValueNode>>::try_ld_set(value)
    }
}
```

Codice DF: Implementazione di `LDConvertSingle` per il tipo `LengthUnit`.

4.8.2. Realizzazione di un unico scope condiviso

Per fare in modo che tutti i tipi necessari fossero in scope, soddisfacendo il requisito indicato in [Sezione 4.7.4.], si é inserito uno statement `pub use …::*` sotto la definizione di ogni modulo contenente un tipo JSON-LD rappresentato, e invece all'interno dei singoli moduli uno statement `use crate::…::*`.

```rust
// ...

mod accept;
pub use accept::*;

mod activity;
pub use activity::*;

mod add;
pub use add::*;

mod announce;
pub use announce::*;

// ...
```

Codice DG: Esempio di utilizzo degli statement `pub use …::*` nel modulo padre per portare in scope tutti i contenuti dei figli.

```rust
use crate::vocabs::*;
use acrate_jsonld_macros::vocab;
```

Codice DH: Esempio di utilizzo degli statement `use crate::…::*` nei moduli figli per portare in scope tutto il contenuto del modulo padre.

4.8.3. Definizione delle strutture dati

Si sono poi ovviamente definite tutte le struct relative ai tipi di ActivityStreams, di cui si riportano alcuni esempi nelle seguenti figure.

```rust
#[vocab]
pub struct Blank {}
```

Codice DI: Definizione del vocab `Blank`, vuoto.

```
#[vocab]
#[inherits(Blank)]
pub struct Entity {
    #[iri = "https://www.w3.org/ns/activitystreams#mediaType"]
    media_type: Option<acrate_jsonld::mediatype::MediaTypeBuf>,

    #[iri = "https://www.w3.org/ns/activitystreams#name"]
    names: Vec<acrate_jsonld::json_ld::LangString>,

    #[iri = "https://www.w3.org/ns/activitystreams#preview"]
    previews: Vec<Link<'src, Source>>,
}
```

Codice DJ: Definizione del vocab `Entity`, contenente le proprietà condivise tra `Object` e `Link`. Si osserva come `previews` sia a sua volta un `Vec` di `Link`, un'altra `struct` taggata con `#[vocab]`.

```
#[vocab]
#[iri = "https://www.w3.org/ns/activitystreams#Object"]
#[inherits(Entity)]
pub struct Object {
    #[iri = "https://www.w3.org/ns/activitystreams#attachment"]
    attachments: Vec<Entity<'src, Source>>,

    #[iri = "https://www.w3.org/ns/activitystreams#attributedTo"]
    attributed_to: Vec<Entity<'src, Source>>,

    #[iri = "https://www.w3.org/ns/activitystreams#audience"]
    audiences: Vec<Entity<'src, Source>>,

    #[iri = "https://www.w3.org/ns/activitystreams#content"]
    contents: Vec<acrate_jsonld::json_ld::LangString>,

    /* ... */

    #[iri = "https://www.w3.org/ns/activitystreams#bto"]
    bto: Vec<Entity<'src, Source>>,

    #[iri = "https://www.w3.org/ns/activitystreams#cc"]
    cc: Vec<Entity<'src, Source>>,

    #[iri = "https://www.w3.org/ns/activitystreams#bcc"]
    bcc: Vec<Entity<'src, Source>>,

    #[iri = "https://www.w3.org/ns/activitystreams#duration"]
    duration: Option<speedate::Duration>,
}
```

Codice DK: Definizione parziale del vocab `Object`. Si osserva che i tipi `LangString` e `Duration` sono stati specificati con percorso globale.

```
#[vocab]
#[iri = "https://www.w3.org/ns/activitystreams#IntransitiveActivity"]
#[inherits(Object)]
pub struct IntransitiveActivity {
    #[iri = "https://www.w3.org/ns/activitystreams#actor"]
    actors: Vec<Entity<'src, Source>>,

    #[iri = "https://www.w3.org/ns/activitystreams#object"]
    objects: Vec<Entity<'src, Source>>,

    #[iri = "https://www.w3.org/ns/activitystreams#target"]
    targets: Vec<Entity<'src, Source>>,

    #[iri = "https://www.w3.org/ns/activitystreams#result"]
    results: Vec<Entity<'src, Source>>,

    #[iri = "https://www.w3.org/ns/activitystreams#origin"]
    origins: Vec<Entity<'src, Source>>,

    #[iri = "https://www.w3.org/ns/activitystreams#instrument"]
    instruments: Vec<Entity<'src, Source>>,
}
```

Codice DL: Definizione del vocab `IntransitiveActivity`, da cui ereditano alcuni tipi di ActivityStreams.

```
#[vocab]
#[iri = "https://www.w3.org/ns/activitystreams#IntransitiveActivity"]
#[inherits(IntransitiveActivity)]
pub struct Activity {
    #[iri = "https://www.w3.org/ns/activitystreams#object"]
    objects: Vec<Entity<'src, Source>>,
}
```

Codice DM: Definizione del vocab `Activity`, da cui ereditano la gran parte dei tipi di ActivityStreams rimanenti, aggiungendo solo raramente altre proprietà.

4.9. Selezione di una licenza

Per il progetto, si é scelto di fare uso della licenza software «European Union Public Licence» [118], in quanto copyleft, conforme alle leggi europee, avente un testo multilingue con valore legale [119] e compatibile con la maggior parte delle licenze copyleft usate nel mondo dell'open source [120], facilitando così il riutilizzo del codice in altri progetti.

5. Conclusioni e sviluppi futuri

Ricapitolando, in questa tesi si é realizzato un set di librerie che permettono di fare uso in programmi Rust [Sezione 3.1.] dei formati di interscambio dati *resource descriptor* [Sezione 4.3.], *NodeInfo* [Sezione 4.5.], *JSON-LD* [Sezione 4.6.], e *ActivityStreams* [Sezione 4.8.], in aggiunta ad un servizio web che genera dinamicamente *resource descriptor* in base alle informazioni contenute in un database [Sezione 4.4.], e una macro che genera automaticamente implementazioni di qualsiasi formato di basato su *JSON-LD* [Sezione 4.7.].

Si é soddisfatti del lavoro svolto, in quanto si ritiene che il software realizzato riesca efficacemente nel suo obiettivo di ridurre la complessità coinvolta nell'utilizzo delle tecnologie implementate.

Benché ci si sia avvicinati significativamente, la realizzazione effettiva della federazione su NextPyter é però ancora mancante: se ne propone una realizzazione futura con ActivityPub [109], il protocollo di social networking utilizzato dai software citati nell'introduzione implementanti il modello federato, i cui prerequisiti sono le tecnologie trattate in questa tesi.

Durante lo sviluppo, si sono scritti unit test [121] per ACRATE, in modo da verificarne il corretto funzionamento anche in assenza di un utilizzatore concreto, potendo inoltre osservare in questo modo le capacità delle librerie realizzate, incluse quelle di supporto al programmatore; a questi potrebbero essere aggiunti unit test che verificano la corretta elaborazione di tutti gli esempi inclusi negli standard, per garantire l'assenza di differenze accidentali di implementazione.

É stato possibile constatare l'efficienza delle librerie solo qualitativamente, osservando il tempo necessario all'esecuzione dei test che si sono definiti, che elaborano deliberatamente degli edge case e quindi non molto rappresentativi di un uso comune; per ottenere misure quantitative, una volta che si avrà un utilizzatore delle librerie, si potranno definire dei benchmark [122] con dei carichi di lavoro campionati dalle elaborazioni effettuate da quest'ultimo.

Sempre attraverso la scrittura di test, si é potuto osservare uno svantaggio della libreria ActivityStreams realizzata: anche se essa é veloce nell'elaborazione dei dati una volta compilata, la compilazione stessa può richiedere molto più tempo di quello che ci si aspetta da una crate Rust, arrivando anche a richiedere 10 secondi. Si suppone che questo sia dovuto alla grande quantità di tipi e implementazioni generati dalla macro per JSON-LD, ma comunque non dovrebbe essere un problema per gli utilizzatori della libreria, in quanto, al di fuori da casi particolari, la dovranno compilare una volta sola.

Infine, si riportano nelle seguenti sezioni quattro ulteriori possibilità di miglioramento di ACRATE, sufficientemente complesse da meritare una breve contestualizzazione.

5.1. Investigazione di un errore di compilazione inaspettato

Scrivendo uno specifico unit test, si é osservata una serie di istruzione inaspettatamente proibite dal compilatore, per la precisione l'accesso ripetuto ma indipendente a proprietà di un proxy di scrittura «`ThingMut`» tramite riferimenti mutabili.

Sarebbe interessante da esplorare, nella possibilità il problema riveli un difetto nella logica di `acrate_jsonld` o, molto più improbabilmente, del compilatore Rust.

```
error[E0499]: cannot borrow `peeeerson_mut` as mutable more than once at a time
   --> acrate_jsonld_macros/tests/test.rs:130:26
    |
124 |        let names = peeeerson_mut
    |                    ------------- first mutable borrow occurs here
...
130 |        let job_titles = peeeerson_mut
    |                         ^^^^^^^^^^^^^
    |                         |
    |                         second mutable borrow occurs here
    |                         first borrow later used here
```

Figura O: Screenshot dell'errore riscontrato, in cui il compilatore Rust segnala che sono stati creati contemporaneamente più riferimenti mutabili a un valore quando in realtà ne esiste solo uno alla volta.

5.2. Pubblicazione come crate separate

Come descritto in [Sezione 4.2.], il progetto é contenuto all'interno di un unico workspace, con le crate che si referenziano tra loro con percorsi interni al workspace attraverso la proprietà `path`.

```
[dependencies]
acrate_database = { path = "../acrate_database", features =
["connect"] }
acrate_rd = { path = "../acrate_rd" }
acrate_utils = { path = "../acrate_utils" }
// ...
```

Codice DN: Un esempio, estratto da `acrate_rdserver` [Sezione 4.4.], di come le crate di ACRATE si referenziano tra loro con `path`.

Idealmente, queste crate andrebbero progressivamente pubblicate su Crates.io, il package registry predefinito [123] di Rust, aggiungendo man mano i parametri `version` alle crate pubblicate usate come dipendenza per permetterne la risoluzione e il recupero anche al di fuori del workspace di ACRATE.

```
[dependencies]
acrate_database = { version = "0.4.0", path = "../acrate_database",
features = ["connect"] }
acrate_rd = { version = "0.4.0", path = "../acrate_rd" }
acrate_utils = { version = "0.4.0", path = "../acrate_utils" }
// ...
```

Codice DO: L'esempio in [Codice DN], con i parametri `version` aggiunti, impostati a `"0.4.0"`, la versione attuale di tutte le crate di ACRATE.

5.3. Aggiunta di funzionalità di documentazione delle implementazioni generate

Al momento, `acrate_jsonld_macros` non permette di specificare docstring per gli item che genera.

Al fine di non avere pagine di documentazione vuote, come in [Figura P], é necessario determinare un modo per includere documentazione negli item generati.

Molto probabilmente, esso convolgerà in qualche modo l'attributo `#[doc]`, funzionalmente equivalente a una docstring `///` o `/** */`.

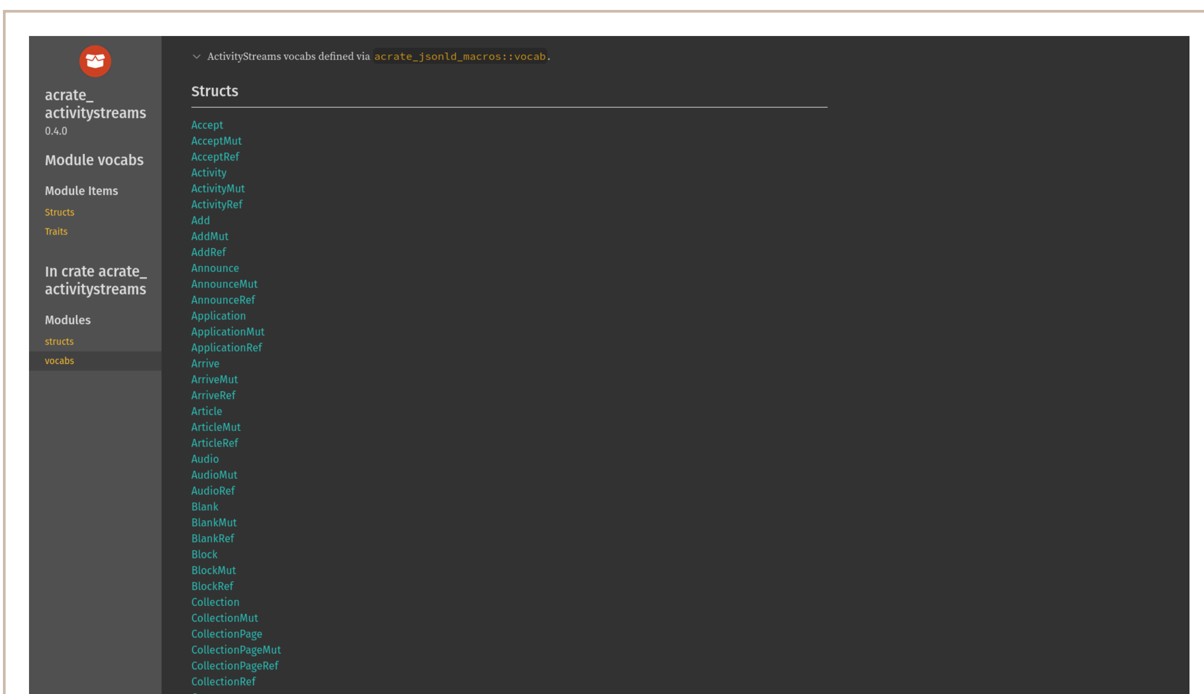

Figura P: Screenshot della documentazione web generata da `rustdoc` per `acrate_activitystreams`. Si osserva che la seconda colonna dell'elenco delle `struct` disponibili é vuota, in quanto esse non hanno documentazione associata.

5.4. Generazione automatica di implementazioni da schemi JSON-LD

Visto che i tipi JSON-LD [Sezione 3.10.6.] sono descritti in schemi machine-readable, `acrate_jsonld_macros` potrebbe essere forsse estesa per leggere il formato machine-readable e generare automaticamente item per tutti i tipi descritti in uno schema, senza bisogno di definire tutte le `struct` `#[vocab]` manualmente come in [Sezione 4.8.3.].

Figure

Figura A L'icona del progetto, una scatola bianca su sfondo arancione. 5
Figura B Screenshot del pannello Cargo di IntelliJ IDEA, mostrante il workspace Rust ACRATE e le relative crate contenute in esso. 5
Figura C Screenshot della root del repository Git di ACRATE, catturato sull'istanza Forgejo in cui il repository é salvato il 2025-06-23. 34
Figura D Screenshot parziale del commit log di ACRATE, catturato sull'istanza Forgejo in cui il repository é salvato. 34
Figura E Esempio di resource descriptor WebFinger ispirato a Mastodon visualizzato in formato human-friendly mentre é in uso un tema colori chiaro sul browser. 54
Figura F Screenshot degli autocompletamenti suggeriti da IntelliJ nella versione finale di `acrate_activitystreams` come metodi con un prefisso di `get_` di `acrate_activitystreams::vocabs::Link`. L'effetto dei metodi é immediatamente chiaro, e non ci sono suggerimenti superflui, come quelli di `Box<json_ld::Node>`, non necessari all'utilizzo di `Link`. 56
Figura G Screenshot degli autocompletamenti suggeriti da IntelliJ come metodi di `Box<json_ld::Node>`. 57
Figura H Diagramma mostrante la struttura del proxy `Thing` con proprietà della sorgente. 61
Figura I Diagramma mostrante la struttura del proxy `ThingRef` con riferimento immutabile alla sorgente. 62
Figura J Diagramma mostrante la struttura del proxy `ThingMut` con riferimento mutabile alla sorgente. 62
Figura K Diagramma dei tipi in input e output di `try_ld_set`. 68
Figura L Diagramma dei tipi in input e output di `try_ld_ref`. 69
Figura M Diagramma dei tipi in input e output di `try_ld_mut`. 70
Figura N Gerarchia finale delle conversioni implementate in `acrate_jsonld::impls::json_ld`. I blocchi blu implementano `LDConvert` direttamente, mentre quelli verdi attraverso `LDConvertSingle`. I blocchi tratteggiati sono inclusi solo se la rispettiva dipendenza é attivata attraverso le feature della crate. 79
Figura O Screenshot dell'errore riscontrato, in cui il compilatore Rust segnala che sono stati creati contemporaneamente più riferimenti mutabili a un valore quando in realtà ne esiste solo uno alla volta. 95
Figura P Screenshot della documentazione web generata da `rustdoc` per `acrate_activitystreams`. Si osserva che la seconda colonna dell'elenco delle `struct` disponibili é vuota, in quanto esse non hanno documentazione associata. 96

Listati

Codice A La variabile `variabile` ha proprietà della stringa creata, e la stringa verrà deallocata al termine della funzione `main`. . . . 8
Codice B La variabile `mutabile` é un riferimento mutabile alla stringa creata. . . . 9
Codice C La variabile `immutabile` é un riferimento immutabile alla stringa creata. . . 9
Codice D Esempio di definizione di un trait e implementazione su una struct. . . . 10
Codice E Estensione dell'esempio di [Codice D] a cui si é aggiunto l'associated type `Errore`. . . . 11
Codice F Definizione di una funzione macro dichiarativa di esempio con `macro_rules!`. . . . 13
Codice G Una chiamata alla funzione macro dichiarativa `Vec`. . . . 13
Codice H Definizione di un attributo macro procedurale di esempio con `#[proc_macro_attribute]`. . . . 13
Codice I Un utilizzo dell'attributo macro procedurale definito in [Codice H]. . . . 14
Codice J Definizione di una struttura dati con supporto a serializzazione e deserializzazione attraverso `serde`. . . . 14
Codice K Definizione di un `enum` rappresentante un errore di esempio che potrebbe essere causato da «questa cosa» o da «quella cosa», con un riferimento alla «cosa» che l'ha causato nel secondo caso. Il tratto `Debug` della «cosa» nel secondo caso viene usato per rappresentarla nel messaggio di errore. . . . 15
Codice L Esempio di chiamata a `warn!` utilizzata per emettere un avvertimento relativo alla variabile `questo`. . . . 15
Codice M Esempio di configurazione che emette su standard output tutti i messaggi di importanza `debug` o superiore provenienti dalla crate `mycrate`. . . . 16
Codice N Una risorsa di un servizio web. Offre accesso al repository di codice di Forgejo. . . . 16
Codice O Header `Accept` inviato da Zen Browser per richiedere pagine web. Manifesta una preferenza per HTML, seguita da XHTML, poi XML, e infine qualsiasi altro formato supportato. . . . 16
Codice P Esempio di template che `minijinja` riporta nella sua documentazione. . . . 18
Codice Q Esempio di inizializzazione di un `minijinja::Environment` che `minijinja` riporta nella sua documentazione. Il primo parametro di `add_template`, `"hello"`, é il nome con cui quel template sarà referenziabile all'interno di altri template, mentre il secondo parametro é il contenuto non valutato del template stesso. . . . 18
Codice R cURL utilizzato per inviare una richiesta HTTPS GET a `example.org`. . . . 18
Codice S Breve Dockerfile non ottimizzato che compila un'applicazione Rust e la rende utilizzabile nell'immagine risultante. . . . 21
Codice T Progetto Docker Compose di esempio per creare l'immagine di un'applicazione ed eseguirla connettendola a un database PostgreSQL istanziato appositamente. . . . 22
Codice U Esempio di utilizzo di `config!` fornito nella documentazione di `micronfig`. Il valore di `MAX_CONCURRENT_USERS` viene convertito a `u64` al momento del

primo accesso, mentre il valore di `SHOWN_ALERT` é una `String` ed é opzionale. 23
Codice V Il file host-meta di `mastodon.social`, che descrive l'URL a cui devono essere effettuate richieste WebFinger. 23
Codice W Il file NodeInfo di `mastodon.social`, recuperato il 2025-06-17. 24
Codice X IRI della pagina di Wiktionary del nome proprio «Rodi» in greco antico. . 25
Codice Y URI equivalente all'IRI in [Codice X]. 25
Codice Z Alcune informazioni riguardanti una persona fittizia, rappresentate in JSON-LD con forma compatta. 26
Codice AA Le stesse informazioni in [Codice Z], rappresentate in forma espansa. Di particolare rilevanza é il fatto che tutte le proprietà, sebbene in forma contratta fossero scalari, siano state trasformate in vettori di cardinalità 1. 26
Codice AB Le stesse informazioni in [Codice Z], rappresentate in forma appiattita. . . 27
Codice AC Documento JSON-LD di esempio con contesto che associa il termine `ste` all'iri `https://meta.steffo.eu/`. 27
Codice AD Esempio di header `Link` impostato per applicare un contesto al documento JSON restituito. 28
Codice AE Documento JSON-LD con la lingua globalmente impostata a Italiano. 28
Codice AF Documento JSON-LD in cui la lingua del valore di `https://meta.steffo.eu/testo` é impostata a Italiano attraverso il contesto. 28
Codice AG Documento JSON-LD privo di contesto in cui la lingua del valore di `https://meta.steffo.eu/testo` é impostata a Italiano direttamente dal valore. 29
Codice AH Documento JSON-LD contenente una language map nella proprietà `https://meta.steffo.eu/testo`, rendendo disponibile il valore di essa sia in Italiano, sia in Inglese (Statunitense). 29
Codice AI Documento JSON-LD di esempio che rappresenta un sonno dalla durata di 9 ore, rappresentata attraverso il tipo XML Schema `duration`. 30
Codice AJ Il valore stringa `"Hello world!"` rappresentato con la crate `json_ld`. 30
Codice AK Documento ActivityStreams rappresentate una breve nota. 31
Codice AL Documento ActivityStreams che descrive l'accettazione di un invito a una festa attraverso un oggetto di tipo `Accept`, che eredita le proprietà `actor` e `object` da `Activity`, i cui valori sono altri oggetti di tipo rispettivamente `Person` e `Invite`, l'ultimo dei quali contiene ancora a sua volta un oggetto `Event` come valore della sua proprietà `object`. 31
Codice AM Definizione semplificata della `struct Accept`. 32
Codice AN Metodi generati su `Accept` per accedere in lettura alla proprietà `summary`. Il primo la legge come una stringa singola, il secondo la legge come una stringa singola con `@language`, il terzo la legge come un vettore di stringhe, e il quarto la legge come un vettore di stringhe con `@language`. 32
Codice AO Il contenuto finale del file `Cargo.toml` che definisce il workspace di ACRATE. 33

Codice AP Esempio di commit message prefissato con il nome della libreria che esso riguarda. 33
Codice AQ Estratto della definizione della `struct` utilizzata per serializzare e deserializzare l'oggetto JSON alla radice dei JSON resource descriptors. 35
Codice AR Implementazione del tratto di conversione `From` che converte la `struct` di `acrate_rd::xrd` descrivente la radice di un resource descriptor alla `struct` equivalente di `acrate_rd::jrd`. Mentre le proprietà `subject` e `aliases` vengono spostate invariate, `properties` viene convertita da un `Vec` a una `HashMap`, e `links`, un vettore di altre `struct` di `acrate_rd::xrd`, viene convertito ricorsivamente a un vettore di `struct` di `acrate_rd::jrd`. 36
Codice AS Breve estratto della funzione `ResourceDescriptorJRD::get` che permette il recupero di un JSON resource descriptor remoto. 37
Codice AT Estratto della definizione dell'`enum` risultante da un errore di recupero di un JSON resource descriptor. 38
Codice AU Estratto della definizione della `struct` rappresentante la radice di un XML resource descriptor. 39
Codice AV Estratto della definizione della `struct` rappresentante il nodo interno `<Link/>` di un XML resource descriptor. 40
Codice AW Estratto della definizione della `struct` rappresentante il nodo interno `<Title>...</Title>` di un XML resource descriptor. 41
Codice AX La definizione di `ResourceDescriptor`. 41
Codice AY Esempio di chiamata alla funzione macro dichiarativa `web_server!` estratto da `acrate_rdserver`, che mostra la sintassi richiesta per specificarne i parametri. Il «parametro» `on:` deve corrispondere all'indirizzo del socket su cui il server deve essere in ascolto per nuove richieste, il «parametro» `templates:` deve corrispondere a una serie di stringhe corrispondenti ai percorsi dei file che devono essere inclusi nel `minijinja::Environment` relativi alla posizione del file di codice in cui si trova la chiamata, e il «parametro» `routes:` deve corrispondere a una serie di coppie risorsa–handler-richiesta separate da `=>`. 42
Codice AZ Definizione della funzione macro dichiarativa `web_server!`. 43
Codice BA Esempio della creazione di una tabella per `acrate_rdserver` tratto da una migrazione. Essa rappresenta un nodo `<Link/>` di un resource descriptor. I tipi stringa in essa usano `BPCHAR` per ignorare gli spazi terminanti. Si é fatto uso di `CONSTRAINT` complessi per verificare alcuni dei vincoli imposti dalla specifica dei resource descriptor: in questo caso, si sta verificando che solo uno tra `href` e `template` non sia NULL. Il prefisso `meta` del nome della tabella deriva da `host-meta`, il nome del file contenente il resource descriptor per un dominio. 44
Codice BB La stessa tabella creata in [Codice BA], rappresentata da una macro ad utilizzo interno di `diesel`. 44

Codice BC Estratto di codice della struct «in lettura» corrispondente alla tabella meta_link_properties. Si può vedere che é presente il parametro id, e che implementa fra le altre cose Queryable. 45
Codice BD Estratto di codice della struct «in scrittura» della stessa tabella di [Codice BC]. Si nota che é assente il parametro id, in quanto selezionato da PostgreSQL al momento della creazione del record, e che implementa Insertable. 45
Codice BE Il Dockerfile completo di acrate_docker. Si può osservare che le prime istruzioni COPY fanno uso del parametro --from=source, nonostante non sia definita un'immagine source. 47
Codice BF Estratto del progetto Docker Compose compose.yml. Si può osservare che la chiave additional_contexts é stata impostata a [source=..] per permettere l'accesso alla cartella immediatamente superiore, corrispondente alla radice del repository. 48
Codice BG Comando per eseguire il setup «di produzione» di acrate_docker. 48
Codice BH Comando per eseguire il setup «di sviluppo» di acrate_docker. 48
Codice BI Configurazione di acrate_database. La variabile ACRATE_DATABASE_URL specifica a quale database l'applicazione dovrà connettersi. 49
Codice BJ Configurazizone di acrate_rdserver. La variabile ACRATE_RDSERVER_BIND_ADDRESS specifica il socket a cui l'applicazione dovrà essere in ascolto per la ricezione di richieste HTTP. 49
Codice BK Progetto Docker Compose di esempio, in cui i container uno e due vengono configurati con lo stesso ACRATE_DATABASE_URL e RUST_LOG, nonostante i loro valori siano scritti una volta sola. Ciò avviene tramite l'anchor YML env, dichiarata in uno e poi referenziata in due. Questo é possibile solo perché entrambi i container usano la stessa chiave per configurare la stessa funzione. 49
Codice BL Progetto Docker Compose di esempio in cui si fa la stessa cosa di [Codice BK] attraverso l'utilizzo dello stesso file .env esterno per entrambi i container. 50
Codice BM Progetto Docker Compose di esempio simile a quello di [Codice BK] in cui nonostante i due eseguibili condividano la stessa chiave di configurazione, essa é configurata a un valore diverso per ciascun container. 50
Codice BN Definizione della funzione healthcheck_handler. 51
Codice BO L'istruzione Dockerfile utilizzata per configurare l'healthcheck nell'immagine Docker di acrate_rdserver. 51
Codice BP Estratto del progetto Docker Compose di sviluppo di ACRATE. Il servizio migrate, che esegue acrate_database_migrate, non viene avviato fino a quando il servizio database non é pronto a ricevere connessioni, mentre il servizio rdserver, che esegue acrate_rdserver, non viene avviato fino a quando il database non é pronto **E** le migrazioni sono state applicate con successo. 52

Codice BQ Esempio di come si sarebbe potuta realizzare ereditarietà multipla vincolando il `trait` `OrderedCollectionPage` a poter essere implementato solo se `OrderedCollection` e `CollectionPage` fossero stati implementati sull'implementatore di `OrderedCollectionPage`. 57
Codice BR Esempio di come si sarebbe potuta realizzare una `struct` con tutte le proprietà del tipo JSON-LD rappresentato. 58
Codice BS Esempio del pattern newtype in cui il nuovo tipo ha la proprietà dell'oggetto contenuto. 58
Codice BT Esempio di implementazione metodi di accesso alle proprietà direttamente sul newtype. 58
Codice BU Esempio semplificato in cui si implementano `trait` di accesso alle proprietà, postfissati con `T`, alle relative `struct`. 59
Codice BV Esempio in cui si rende generico il tipo `OrderedCollectionPage`. 60
Codice BW Il `trait` `Source` di [Codice BV] dopo la triplicazione dei proxy. 62
Codice BX Definizione del tipo `LDKey`. 63
Codice BY Definizione del tratto `LDValue`. Gli associated type `Ref` e `Mut` sono vincolati ad avere un lifetime di almeno `'src`. 63
Codice BZ Implementazione di `LDValue` per il tipo `IndexedObject` della crate `json_ld`. Come da vincolo, `Ref` e `Mut` hanno un lifetime di `'src`. 63
Codice CA Ipotetica implementazione di `LDValue` per `String`, per una ipotetica source `HashMap<String, String>`. Si evidenzia come i tipi di `Ref` e `Mut` siano diversi (`str`) dal tipo implementatore (`String`)! 64
Codice CB Estratto della definizione di `LDSource`, in cui si evidenziano `LDKey` e `LDValue`. 64
Codice CC Definizione di `LDTypes`. 65
Codice CD Definizione di `LDMutTypes`. 65
Codice CE Estratto della definizione di `AssociatedLDTypes`. `associated_ld_types` restituisce un `HashSet` di proprietà nonostante sia un metodo statico per via di problemi inesplorati riscontrati durante la sua realizzazione con un riferimento immutabile statico. 66
Codice CF L'*implementazione a tappeto* di `AssociatedLDMutTypes`. 66
Codice CG Estratto della definizione del trait `LDConvert`, che mostra il tipo generico `Value`. 67
Codice CH Estratto della definizione del trait `LDConvert`, che mostra il metodo `try_ld_set` e i relativi tipi associati. 68
Codice CI Estratto della definizione del trait `LDConvert`, che mostra il metodo `try_ld_ref` e i relativi tipi associati. 69
Codice CJ Estratto della definizione del trait `LDConvert`, che mostra il metodo `try_ld_ref` e i relativi tipi associati. 70
Codice CK Definizione completa di `LDConvert`. 71
Codice CL Estratto della definizione del metodo `LDSource::get_ld_item`. 72
Codice CM Estratto della definizione dei metodi `LDMutSource::get_ld_item_mut` e `LDMutSource::set_ld_item`. 73

Codice CN Estratto della definizione di `LDConvertSingle`, che evidenzia ccome sono cambiati i metodi `try_ld_ref`, `try_ld_mut` e `try_ld_set` da `LDConvert`. . 74

Codice CO Definizione del sottomodulo `json_ld`. L'attributo `#[cfg(feature = "json-ld")]` lo include nella build solamente se la feature con quel nome é abilitata, rendendo così la crate `json_ld` una dipendenza opzionale. 74

Codice CP L'implementazione di `LDConvert` per `Vec<IndexedObject>`. I metodi non effettuano nessuna elaborazione, e si limitano a restituire l'input all'interno di un risultato `Ok`. Si può notare come il tipo di errore utilizzato sia `Infallible`, un tipo speciale di cui il compilatore sa che non possono esistere istanze, e quindi ottimizza via. 75

Codice CQ Estratto della definizione di `ValueNode`, confrontato con un estratto della definizione di `json_ld::Object`. Si osserva che `ValueNode` non prevede più la variante `List`. 75

Codice CR L'implementazione di `LDConvert` per `Vec<T>`, dove `T` é qualsiasi tipo implementante `LDConvertSingle`. 77

Codice CS Estratto dell'implementazione di `LDConvert` per `Option<T>`, dove `T` é qualsiasi tipo implementante `LDConvertSingle`. Si sono aggiunti dei warning per le build di debug nel caso in cui il tipo letto come `Option<T>` contenga più di un valore. 78

Codice CT Chiamata alla macro `vocab!` in una vecchia versione del codice in cui molte delle astrazioni descritte non esistevano ancora. Il tipo `Activity` ha solo i metodi specifici alla propria proprietà `object`, e non quelli delle proprietà dei tipi da cui eredita, come `subject`. 80

Codice CU Rappresentazione di esempio del tipo JSON-LD `Person` con solo la proprietà `name` inclusa. É utilizzato per effettuare testing di `acrate_jsonld_macros`. 81

Codice CV Definizione di `BaseBlocksBuilder`. Il tipo `proc_macro2::TokenStream` é chiamato `TokenStream2` per distinguerlo dal tipo della standard library `proc_macro::TokenStream`, di cui non si fa utilizzo. 82

Codice CW Definizione della funzione `vec_from_data`, che riceve in input i token della `struct` dichiarata, ed emette in output un `Vec` di `FieldConfig` che `BaseBlocksBuilder::add_field` é in grado di elaborare. Questo pezzo di codice nello specifico verifica che l'item a cui é stato aggiunto l'attributo sia una `struct` con campi dal nome esplicito, e non un `enum`, una `union`, una `struct` vuota, o una `struct` con campi posizionali, delegando poi il resto dell'elaborazione a `FieldConfig::from_field`. 83

Codice CX La signature di `add_field`. 83

Codice CY Estratto di codice della funzione `add_field`, che mostra l'aggiunta di un blocco di codice al `Vec reader_impl`. Il codice é stato ri-indentato per renderlo più facilmente leggibile. Gli identificatori prefissati con # sono sostituzioni: ad esempio, `#ty` viene sostituito nel codice generato con il tipo del campo a cui il metodo corrisponde. Ad esempio, in [Codice CU], implementando il campo `names`, `#ty` verrebbe sostituito con `Vec<&'src`

str>, inserendo così una chiamata a get_ld_item a cui é imposto di restituire Vec<&'src str>. 84
Codice CZ Breve estratto del template utilizzato per convertire BaseBlocksBuilder in un unico TokenStream. #(#reader_trait)* effettua il concatenamento del Vec reader_trait, poi lo usa conme contenuto del blocco di definizione del trait ThingReader, dove Thing é il nome del tipo JSON-LD. 84
Codice DA Definizione della variabile statica INHERIT_STORAGE, il cui contenuto é condiviso tra tutte le chiamate alla macro vocab. 85
Codice DB Definizione della struct InheritStorage, condivisa tra struct taggate con #[vocab]. 86
Codice DC La definizione completa della funzione generate_recursion. 87
Codice DD Definizione del tipo LengthUnit, usato per la proprietà unit del tipo ActivityStreams Place. 88
Codice DE Implementazione di From<String> per il tipo LengthUnit. 89
Codice DF Implementazione di LDConvertSingle per il tipo LengthUnit. 89
Codice DG Esempio di utilizzo degli statement pub use …::* nel modulo padre per portare in scope tutti i contenuti dei figli. 90
Codice DH Esempio di utilizzo degli statement use crate::…::* nei moduli figli per portare in scope tutto il contenuto del modulo padre. 90
Codice DI Definizione del vocab Blank, vuoto. 90
Codice DJ Definizione del vocab Entity, contenente le proprietà condivise tra Object e Link. Si osserva come previews sia a sua volta un Vec di Link, un'altra struct taggata con #[vocab]. 91
Codice DK Definizione parziale del vocab Object. Si osserva che i tipi LangString e Duration sono stati specificati con percorso globale. 92
Codice DL Definizione del vocab IntransitiveActivity, da cui ereditano alcuni tipi di ActivityStreams. 93
Codice DM Definizione del vocab Activity, da cui ereditano la gran parte dei tipi di ActivityStreams rimanenti, aggiungendo solo raramente altre proprietà. . 93
Codice DN Un esempio, estratto da acrate_rdserver [Sezione 4.4.], di come le crate di ACRATE si referenziano tra loro con path. 95
Codice DO L'esempio in [Codice DN], con i parametri version aggiunti, impostati a "0.4.0", la versione attuale di tutte le crate di ACRATE. 96

Tabelle

Tabella A Schema delle tabelle utilizzate da `acrate_rdserver`. Più informazioni su come esse vengono interpretate sono fornite in [Sezione 4.4.7.]. 46

Tabella B Documento inviato da `acrate_rdserver` in corrispondenza di ciascun formato. 53

Tabella C I cinque item Rust definiti per rappresentare un tipo JSON-LD, dove il nome del tipo é indicato come `Thing`. 61

Bibliografia

[1] *Regolamento (UE) 2016/679 del Parlamento Europeo e del Consiglio, del 27 aprile 2016, relativo alla protezione delle persone fisiche con riguardo al trattamento dei dati personali, nonché alla libera circolazione di tali dati e che abroga la direttiva 95/46/CE (regolamento generale sulla protezione dei dati).* Unione Europea, 2016. [Online]. Disponibile su: **https:**//eur-lex.europa.eu/legal-content/IT/TXT/HTML/?uri=CELEX:32016R0679#tit_1

[2] «Digital Sovereignty for Europe», European Parliament, Ideas Paper. [Online]. Disponibile su: **https:**//www.europarl.europa.eu/RegData/etudes/BRIE/2020/651992/EPRS_BRI(2020)651992_EN.pdf

[3] «FAQs», Interoperable Europe Portal. [Online]. Disponibile su: **https:**//interoperable-europe.ec.europa.eu/interoperable-europe/faqs

[4] Stefano Pigozzi, «Progettazione e sviluppo di Sophon, applicativo cloud a supporto della ricerca». [Online]. Disponibile su: **https:**//gh.steffo.eu/sophon/index.html

[5] Stefano Pigozzi, Francesco Faenza, e Claudia Canali, «Sophon: an Extensible Platform for Collaborative Research», in *Practice and Experience in Advanced Research Computing 2022: Revolutionary: Computing, Connections, You*, Association for Computing Machinery, lug. 2022. **doi:** 10.1145/3491418.3535163.

[6] Francesco Faenza, Claudia Canali, Lisa Fregni, e Emiliano Maccaferri, «NextPyter: Open-Source Research Collaborative Platform», in *PEARC '24: Practice and Experience in Advanced Research Computing 2024: Human Powered Computing*, Association for Computing Machinery, lug. 2024. **doi:** 10.1145/3626203.3670516.

[7] Francesco Faenza, Emiliano Maccaferri, e Claudia Canali, «Containerized Jupyter Notebooks: balancing flexibility and performance», 24 ottobre 2024, *Institute of Electrical and Electronics Engineers*. **doi:** 10.1109/NCA61908.2024.00033.

[8] «FediDB: The Complete Fediverse Directory & Analytics Platform». [Online]. Disponibile su: **https:**//fedidb.com/

[9] «Mastodon - Decentralized social media». [Online]. Disponibile su: **https:**//joinmastodon.org/

[10] «ec.social-network.europa.eu». [Online]. Disponibile su: **https:**//ec.social-network.europa.eu/about

[11] «Open Science». [Online]. Disponibile su: **https:**//bonfirenetworks.org/app/open-science/

[12] «Encyclia.pub». [Online]. Disponibile su: **https:**//encyclia.pub/

[13] «Forgejo monthly update - January 2025». [Online]. Disponibile su: **https:**//forgejo.org/2025-01-monthly-update/

[14] «IETF | RFCs». [Online]. Disponibile su: **https:**//www.ietf.org/process/rfcs/

[15] «WebFinger», feb. 2023. [Online]. Disponibile su: https://docs.joinmastodon.org/spec/webfinger/

[16] «Web Standards | W3C». [Online]. Disponibile su: https://www.w3.org/standards/

[17] «ActivityPub», mag. 2025. [Online]. Disponibile su: https://docs.joinmastodon.org/spec/activitypub/

[18] «Rust Programming Language». Consultato: 9 giugno 2025. [Online]. Disponibile su: https://www.rust-lang.org/

[19] «Understanding Ownership», in *The Rust Programming Language*. Consultato: 9 giugno 2025. [Online]. Disponibile su: https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html

[20] «What is Ownership?», in *The Rust Programming Language*. Consultato: 9 giugno 2025. [Online]. Disponibile su: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

[21] «References and Borrowing», in *The Rust Programming Language*. Consultato: 9 giugno 2025. [Online]. Disponibile su: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

[22] «Aliasing», in *The Rustonomicon*. Consultato: 9 giugno 2025. [Online]. Disponibile su: https://doc.rust-lang.org/nomicon/aliasing.html

[23] «Networking», Rust Programming Language. Consultato: 9 giugno 2025. [Online]. Disponibile su: https://www.rust-lang.org/what/networking

[24] *WebRender*. Consultato: 9 giugno 2025. [Online]. Disponibile su: https://github.com/servo/webrender/tree/c4bd5b47d8f5cd684334b445e67a1f945d106848

[25] *Servo*. Consultato: 9 giugno 2025. [Online]. Disponibile su: https://github.com/servo/servo/tree/7f536e8092a098ba809252033954fe2120b4b028

[26] Jesse Howarth, «Why Discord is switching from Go to Rust». Consultato: 9 giugno 2025. [Online]. Disponibile su: https://discord.com/blog/why-discord-is-switching-from-go-to-rust

[27] «1Password». Consultato: 9 giugno 2025. [Online]. Disponibile su: https://rustfoundation.org/members/

[28] «Sentry». Consultato: 9 giugno 2025. [Online]. Disponibile su: https://rustfoundation.org/members/

[29] Matt Asay, «Why AWS loves Rust, and how we'd like to help». [Online]. Disponibile su: https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/

[30] «impl<T: ?Sized> Copy for &T». [Online]. Disponibile su: https://doc.rust-lang.org/stable/std/marker/trait.Copy.html#impl-Copy-for-%26T

[31] «Validating References with Lifetimes», in *The Rust Programming Language*. Consultato: 10 giugno 2025. [Online]. Disponibile su: **https:**//doc.rust-lang.org/book/ch10-03-lifetime-syntax.html

[32] «Packages and Crates», in *The Rust Programming Language*. Consultato: 10 giugno 2025. [Online]. Disponibile su: **https:**//doc.rust-lang.org/book/ch07-01-packages-and-crates.html

[33] «Traits: Defining Shared Behavior», in *The Rust Programming Language*. Consultato: 10 giugno 2025. [Online]. Disponibile su: **https:**//doc.rust-lang.org/book/ch10-02-traits.html

[34] «impl<T: PartialEq, A: Allocator> for Vec<T, A>». [Online]. Disponibile su: **https:**//doc.rust-lang.org/stable/std/vec/struct.Vec.html#impl-Vec%3CT,+A%3E-2

[35] «Associated Items», *The Rust Standard Library*. [Online]. Disponibile su: **https:**//doc.rust-lang.org/reference/items/associated-items.html

[36] «Trait FromStr». [Online]. Disponibile su: **https:**//doc.rust-lang.org/stable/std/str/trait.FromStr.html

[37] «re_rebalancing_coherence». [Online]. Disponibile su: **https:**//rust-lang.github.io/rfcs/2451-re-rebalancing-coherence.html#concrete-orphan-rules

[38] «Items», in *The Rust Reference*. [Online]. Disponibile su: **https:**//doc.rust-lang.org/reference/items.html

[39] *The Cargo Book*. Consultato: 10 giugno 2025. [Online]. Disponibile su: **https:**//doc.rust-lang.org/cargo/index.html

[40] «The Manifest Format», *The Rust Standard Library*. [Online]. Disponibile su: **https:**//doc.rust-lang.org/nightly/cargo/reference/manifest.html

[41] «Features», in *The Cargo Book*. Consultato: 24 giugno 2025. [Online]. Disponibile su: **https:**//doc.rust-lang.org/cargo/reference/features.html

[42] «Macros», in *The Rust Programming Language*. Consultato: 11 giugno 2025. [Online]. Disponibile su: **https:**//doc.rust-lang.org/book/ch20-05-macros.html

[43] «Procedural Macros», *The Rust Standard Library*. [Online]. Disponibile su: **https:**//doc.rust-lang.org/reference/procedural-macros.html

[44] «Macros By Example», in *The Rust Reference*. [Online]. Disponibile su: **https:**//doc.rust-lang.org/reference/macros-by-example.html

[45] «macro_rules!». [Online]. Disponibile su: **https:**//doc.rust-lang.org/rust-by-example/macros.html

[46] «Macro vec». [Online]. Disponibile su: **https:**//doc.rust-lang.org/std/macro.vec.html

[47] «serde». [Online]. Disponibile su: **https:**//docs.rs/serde/latest/serde/

[48] «Using derive». [Online]. Disponibile su: https://serde.rs/derive.html

[49] Stefano Pigozzi, «Analisi su grafo Neo4J relativo alle dipendenze delle crates del linguaggio Rust», giu. 2023. [Online]. Disponibile su: https://forge.steffo.eu/unimore/bda-5-steffo/src/branch/main/README.md

[50] «thiserror». [Online]. Disponibile su: https://docs.rs/thiserror/latest/thiserror/

[51] «log». [Online]. Disponibile su: https://docs.rs/log/latest/log/

[52] «pretty_env_logger». [Online]. Disponibile su: https://docs.rs/pretty_env_logger/latest/pretty_env_logger/

[53] «Web Services Architecture», W3C, Working Group Note, feb. 2004. [Online]. Disponibile su: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

[54] «Content negotiation». [Online]. Disponibile su: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Content_negotiation

[55] «axum». [Online]. Disponibile su: https://docs.rs/axum/latest/axum/

[56] «axum::extract». [Online]. Disponibile su: https://docs.rs/axum/latest/axum/extract/index.html

[57] «FromRequest in axum::extract». [Online]. Disponibile su: https://docs.rs/axum/latest/axum/extract/trait.FromRequest.html

[58] «Extension in axum». [Online]. Disponibile su: https://docs.rs/axum/latest/axum/struct.Extension.html

[59] «IntoResponse in axum::response». [Online]. Disponibile su: https://docs.rs/axum/latest/axum/response/trait.IntoResponse.html

[60] «axum::response». [Online]. Disponibile su: https://docs.rs/axum/latest/axum/response/index.html

[61] «Welcome to Flask». [Online]. Disponibile su: https://flask.palletsprojects.com/en/stable/

[62] «minijinja». [Online]. Disponibile su: https://docs.rs/minijinja/latest/minijinja/

[63] «FAQ -- Frequently Asked Questions». [Online]. Disponibile su: https://curl.se/docs/faq.html#What_is_cURL

[64] «curl - How To Use». [Online]. Disponibile su: https://curl.se/docs/manpage.html

[65] Daniel Stenberg, «My name appears in products». [Online]. Disponibile su: https://daniel.haxx.se/my-name-in-products.html

[66] «Extensible Markup Language (XML)», W3C, ott. 2016. [Online]. Disponibile su: https://www.w3.org/XML/

[67] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, e François Yergeau, A c. di, «Extensible Markup Language (XML) 1.0 (Fifth Edition)». W3C, 26 novembre 2008. [Online]. Disponibile su: https://www.w3.org/TR/xml/

[68] Don Box *et al.*, «Simple Object Access Protocol (SOAP) 1.1», W3C, Note, mag. 2000. [Online]. Disponibile su: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[69] «quick_xml». [Online]. Disponibile su: https://docs.rs/quick-xml/latest/quick_xml/

[70] «quick_xml::de». [Online]. Disponibile su: https://docs.rs/quick-xml/latest/quick_xml/de/index.html

[71] «Introducing JSON». [Online]. Disponibile su: https://www.json.org/json-en.html

[72] «JSON - JavaScript». [Online]. Disponibile su: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

[73] «serde_json». [Online]. Disponibile su: https://docs.rs/serde_json/latest/serde_json/

[74] «PostgreSQL: About». [Online]. Disponibile su: https://www.postgresql.org/about/

[75] «Diesel is a Safe, Extensible ORM and Query Builder for Rust». [Online]. Disponibile su: https://diesel.rs/

[76] «Getting Started with Diesel». [Online]. Disponibile su: https://diesel.rs/guides/getting-started.html

[77] «diesel_async». [Online]. Disponibile su: https://docs.rs/diesel-async/latest/diesel_async/

[78] «What is Docker?». [Online]. Disponibile su: https://docs.docker.com/get-started/docker-overview/

[79] «What is an image?». [Online]. Disponibile su: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/

[80] *rust - Official Image*. [Online]. Disponibile su: https://hub.docker.com/_/rust

[81] *debian - Official Image*. [Online]. Disponibile su: https://hub.docker.com/_/debian

[82] «Writing a Dockerfile». [Online]. Disponibile su: https://docs.docker.com/get-started/docker-concepts/building-images/writing-a-dockerfile/

[83] «What is a container?». [Online]. Disponibile su: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/

[84] «What is Docker Compose?». [Online]. Disponibile su: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-docker-compose/

[85] Stefano Pigozzi, «micronfig». [Online]. Disponibile su: https://docs.rs/micronfig/latest/micronfig/

[86] Blaine Cook e Eran Hammer-Lahav, «Web Host Metadata», RFC Editor, Request for Comments 6415, ott. 2011. doi: 10.17487/RFC6415.

[87] Joe Hildebrand, Peter Saint-Andre, e Lance Stout, «Discovering Alternative XMPP Connection Methods», XMPP Standards Foundation, XMPP Extension Protocol XEP156, feb. 2022. [Online]. Disponibile su: https://xmpp.org/extensions/xep-0156.html

[88] Paul Jones, Gonzalo Salgueiro, Michael B. Jones, e Joseph Smarr, «WebFinger», RFC Editor, Request for Comments 7033, set. 2013. doi: 10.17487/RFC7033.

[89] Nat Sakimura, John Bradley, Michael B. Jones, e Edmund Jay, «OpenID Connect Discovery 1.0 incorporating errata set 2», dic. 2023. [Online]. Disponibile su: https://openid.net/specs/openid-connect-discovery-1_0.html

[90] *NodeInfo*. [Online]. Disponibile su: https://github.com/jhass/nodeinfo

[91] «NodeInfo schema version 2.2». 21 dicembre 2023. [Online]. Disponibile su: https://github.com/jhass/nodeinfo/blob/15b74ef1b7b2c300cf701d3d6c81c72a306d9d24/schemas/2.2/schema.json

[92] «Fediverse Observer». [Online]. Disponibile su: https://fediverse.observer/stats

[93] «the federation - a statistics hub». [Online]. Disponibile su: https://the-federation.info/

[94] «JSON-LD - JSON for Linked Data». [Online]. Disponibile su: https://json-ld.org/

[95] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, Pierre-Antoine Champin, e Niklas Lindström, «JSON-LD 1.1», W3C, Recommendation, lug. 2020. [Online]. Disponibile su: https://www.w3.org/TR/2020/REC-json-ld11-20200716/

[96] «JSON-LD | Gmail | Google for Developers». 4 giugno 2025. [Online]. Disponibile su: https://developers.google.com/workspace/gmail/markup/reference/formats/json-ld

[97] «European data». [Online]. Disponibile su: https://data.europa.eu/data/datasets?locale=en&format=JSON_LD

[98] Martin J. Dürst e Michel Suignard, «Internationalized Resource Identifiers (IRIs)». [Online]. Disponibile su: https://www.rfc-editor.org/rfc/rfc3987.html

[99] Tim Berners-Lee, Roy T. Fielding, e Larry M Masinter, «Uniform Resource Identifier (URI): Generic Syntax», RFC Editor, Request for Comments 3986, gen. 2005. doi: 10.17487/RFC3986.

[100] Addison Phillips e Mark Davis, «Tags for Identifying Languages». [Online]. Disponibile su: https://www.rfc-editor.org/rfc/rfc5646.html

[101] David Peterson, Shudi (Sandy) Gao, Ashok Malhotra, C. M. Sperberg-McQueen, Henry S. Thompson, e Paul V. Biron, A c. di, «W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes», W3C, Recommendation, apr. 2012. [Online]. Disponibile su: https://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/

[102] «json_ld». [Online]. Disponibile su: https://docs.rs/json-ld/latest/json_ld/

[103] «IndexedObject in json_ld». [Online]. Disponibile su: https://docs.rs/json-ld/latest/json_ld/type.IndexedObject.html

[104] «Object in json_ld». [Online]. Disponibile su: https://docs.rs/json-ld/latest/json_ld/enum.Object.html

[105] «Value in json_ld». [Online]. Disponibile su: https://docs.rs/json-ld/latest/json_ld/enum.Value.html

[106] «Literal in json_ld::object». [Online]. Disponibile su: https://docs.rs/json-ld/latest/json_ld/object/enum.Literal.html

[107] «SmallString in smallstr». [Online]. Disponibile su: https://docs.rs/smallstr/0.3.0/smallstr/struct.SmallString.html

[108] James M Snell e Evan Prodromou, A c. di, «Activity Streams 2.0», W3C, Recommendation, mag. 2017. [Online]. Disponibile su: https://www.w3.org/TR/2017/REC-activitystreams-core-20170523/

[109] Christine Lemmer-Webber, Jessica Tallon, Erin Shepherd, Amy Guy, e Evan Prodromou, «ActivityPub», W3C, Recommendation, gen. 2018. [Online]. Disponibile su: https://www.w3.org/TR/2018/REC-activitypub-20180123/

[110] Amy Guy, A c. di, «Social Web Protocols», W3C, Working Group Note, dic. 2017. [Online]. Disponibile su: https://www.w3.org/TR/2017/NOTE-social-web-protocols-20171225/

[111] James M Snell e Evan Prodromou, A c. di, «Activity Vocabulary», W3C, Recommendation, mag. 2017. [Online]. Disponibile su: https://www.w3.org/TR/2017/REC-activitystreams-vocabulary-20170523/

[112] *LemmyNet/lemmy*. [Online]. Disponibile su: https://github.com/LemmyNet/lemmy

[113] *LemmyNet/activitypub-federation-rust*. [Online]. Disponibile su: https://github.com/LemmyNet/activitypub-federation-rust

[114] *asonix/activitystreams*. [Online]. Disponibile su: https://git.asonix.dog/asonix/activitystreams

[115] «activitystreams». [Online]. Disponibile su: https://docs.rs/activitystreams/latest/activitystreams/

[116] «Accept in activitystreams::activity». [Online]. Disponibile su: https://docs.rs/activitystreams/latest/activitystreams/activity/struct.Accept.html

[117] Emiliano Maccaferri e Francesco Faenza, *NextPyter / daemon*. [Online]. Disponibile su: **https:**//gitlab.com/nextpyter/daemon

[118] «EUPL text (EUPL-1.2)», Interoperable Europe Portal. [Online]. Disponibile su: **https:**//interoperable-europe.ec.europa.eu/collection/eupl/eupl-text-eupl-12

[119] «European Union Public License». [Online]. Disponibile su: **https:**//commission.europa.eu/about/departments-and-executive-agencies/digital-services/open-source-strategy-history/european-union-public-licence_en

[120] *Decisione di esecuzione (UE) 2017/863 della Commissione, del 18 maggio 2017, che aggiorna la licenza EUPL per il software con codice sorgente aperto per agevolare ulteriormente la condivisione e il riutilizzo del software sviluppato dalle pubbliche amministrazioni*. Unione Europea, 2017, pp. 59–64. [Online]. Disponibile su: **https:**//eur-lex.europa.eu/eli/dec_impl/2017/863/oj/ita

[121] «cargo-test(1)», in *The Cargo Book*. Consultato: 20 giugno 2025. [Online]. Disponibile su: **https:**//doc.rust-lang.org/cargo/commands/cargo-test.html

[122] «cargo-bench(1)», in *The Cargo Book*. Consultato: 20 giugno 2025. [Online]. Disponibile su: **https:**//doc.rust-lang.org/cargo/commands/cargo-bench.html

[123] «cargo-publish(1)», in *The Cargo Book*. Consultato: 20 giugno 2025. [Online]. Disponibile su: **https:**//doc.rust-lang.org/cargo/commands/cargo-publish.html