ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
DIPARTIMENTO DI
INGEGNERIA DELL'ENERGIA ELETTRICA E
DELL'INFORMAZIONE
"GUGLIELMO MARCONI"

CORSO DI LAUREA IN INGEGNERIA BIOMEDICA

TITOLO DELL'ELABORATO

SEPABot: notifiche su Discord di eventi semantici

Elaborato in
CALCOLATORI ELETTRONICI

Relatore

Luca Roffia

Correlatore

Gregorio Monari

Presentata da

Matteo Vitali

Anno Accademico 2022/2023

Abstract

In questa tesi si vedrà lo sviluppo di un Bot in grado di prelevare dati da un database semantico tramite SEPA (SPARQL Event Processing Architecture), rielaborarli e inviarli a Discord. Lo scopo principale di questo studio è fornire un metodo comodo ed efficiente per il tracciamento del flusso dati all’interno del database.

Dopo una breve introduzione sul funzionamento, verranno esaminati gli strumenti utilizzati e le varie componenti del bot.

Il Bot offre molteplici possibilità di utilizzo. Saranno illustrati casi d'uso come la gestione di messaggi d'errore e di nuovi login.

Indice

Elenco delle figure

0 Introduzione

Avere mezzi di comunicazione efficienti è fondamentale per il successo e l'organizzazione di qualsiasi tipo di attività. Questo principio si applica anche nel contesto dello sviluppo software. Una comunicazione aperta e trasparente permette a tutti i membri del team di essere aggiornati sugli obiettivi comuni e, in questo caso specifico, di monitorare il flusso dati all'interno del database. Inoltre, avere dei buoni mezzi comunicativi è fondamentale anche per le interazioni con gli utenti finali. Una comunicazione chiara e tempestiva con i clienti consente di comprendere meglio le loro esigenze, di rispondere alle loro richieste e di fornire soluzioni che soddisfino le loro aspettative (come si vedrà nell'ultimo capitolo dedicato ai potenziali sviluppi futuri). Pertanto, è fondamentale trovare un canale di comunicazione che sia versatile, comodo e veloce, accessibile sia agli sviluppatori che al cliente. L'utilizzo di un Bot rappresenta un'ottima soluzione per raggiungere questi obiettivi. Grazie a un Bot, è possibile garantire una comunicazione costante, fornendo aggiornamenti in tempo reale sul flusso dei dati, gestendo notifiche, errori e interazioni con gli utenti finali. In più un Bot, può essere personalizzato per adattarsi alle esigenze specifiche del progetto, offrendo una piattaforma centralizzata per la gestione delle comunicazioni.

Il Bot in questione è sviluppato interamente in Python ed è composto da due tipologie di componenti: consumatore e aggregatore. Il consumatore è unico e si occupa di prelevare i dati dal database (SEPA) e inoltrarli a Discord, con una minima rielaborazione. Gli aggregatori invece sono molteplici e la loro funzione è prelevare i dati, rielaborarli, e rimetterli dentro il database nella condizione di essere poi presi dal consumatore. Questa suddivisione consente di lasciare intatto il consumatore e aggiungere tutti gli aggregatori di cui si necessita, favorendo la scalabilità del Bot. Il consumatore può essere configurato comodamente dagli sviluppatori per mandare i dati ai clienti o agli admin. (Per l'analisi della struttura completa si veda il capitolo 2).

Questa struttura a blocchi rende il Bot un canale di comunicazione adatto a soddisfare tutte le caratteristiche elencate sopra, rendendolo anche modificabile ed espandibile da altri sviluppatori (ad esempio un aggregatore può essere costruito anche utilizzando altri linguaggi di programmazione).

0.1 Struttura della tesi

L'elaborato è suddiviso in tre capitoli, escluso il capitolo introduttivo:

- Il primo capitolo è dedicato alle tecnologie di sviluppo. Fornisce i fondamenti teorici necessari a comprendere il funzionamento del Bot.
- Il secondo capitolo è incentrato sullo sviluppo del software e mostra in dettaglio la struttura del Bot. Si vedranno le librerie Python di riferimento e verrà fornita una trattazione dettagliata del concetto di consumatore e aggregatore.
- Il terzo e ultimo capitolo presenta le conclusioni ed espone possibili sviluppi futuri.

1 Tecnologie di sviluppo

In questo capitolo si vedranno tutti gli strumenti impiegati durante lo sviluppo. Saranno esaminati i protocolli e i formati utilizzati per il trasferimento dei dati, nonché le piattaforme adottate per la realizzazione del progetto.

Per garantire una comunicazione efficiente e standardizzata, è stato adottato il protocollo HTTP per la trasmissione delle informazioni. Inoltre, i dati verranno formattati in formato JSON, ampiamente diffuso e facilmente eseguibile da tutti i sistemi più comuni, compresi smart phone e tablet. Per consentire uno scambio rapido ed efficiente con la piattaforma di Discord sarà utilizzata una API (Discord developer interface). Il database che andremo a trattare sarà un database semantico, che risponde al corrispettivo linguaggio di interrogazione SPARQL. Per la configurazione di consumatore, aggregatori e dashboard SEPA ci serviremo di un file JSAP, simile al JSON, nel quale inseriremo tutte le variabili di cui si necessita.

1.1 HTTP e JSON

L' HyperText Transfer Protocol (HTTP) è un protocollo tipicamente utilizzato per la trasmissione di informazioni sul web, ovvero per tutti i flussi di dati che seguono un'architettura di tipo client-server [1].

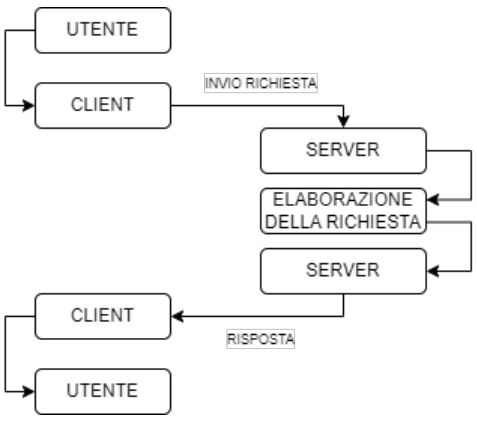

Figura 1. Modello client-server

Ogni richiesta di dato effettuata dal client inizia con un messaggio di testo creato in formato HTTP. Questo è una serie di caratteri organizzati in modo standardizzato. Una richiesta HTTP ha la forma:

[method] [URI] [version]
[header]
[body]

Le componenti più importanti si trovano nella prima riga. Il tag ‘method’ definisce il modo con cui il client vuole interagire con la risorsa, alcune delle operazioni più comuni sono GET e POST che si occupano rispettivamente di richiedere o inviare il dato. L’URI (Universal Resource Identifier) è invece il codice identificativo univoco della risorsa alla quale si vuole accedere. Ogni oggetto che si trova sul web ha un proprio URI e un proprio URL (Universal Resource Locator) che invece rappresenta la sua posizione. L’header invece serve a contenere eventuali informazioni aggiuntive mentre il body contiene il corpo del messaggio (opzionale).

Il server conosce la risorsa richiesta (tramite l’URI) e cosa fare con quella risorsa (tramite il method), dovrà quindi mandare una risposta al client. Come per la richiesta anche la risposta dovrà essere erogata in modo standardizzato, una risposta HTTP ha la seguente forma:

[version] [status code] [reason]
[header]
[body]

Lo status code è un codice di tre cifre che mi informa delle condizioni della richiesta. Ad esempio, i codici che iniziano con il numero due comunicano che l’operazione è stata portata a termine con successo, altri numeri possono significare errori da parte del client, del server o anche di altro tipo.

Il ‘payload’ di un messaggio HTTP, il dato vero e proprio, è una stringa di caratteri. Trasmettere dati sotto forma di stringa è vantaggioso perché convertire bit in caratteri testuali risulta molto semplice. Il formato JSON [2] è un formato di testo in cui le informazioni sono organizzate in modo da potere essere serializzate e deserializzate. In questo modo è possibile trasmettere strutture dati con un linguaggio comprensibile sia ad umani che a computer.

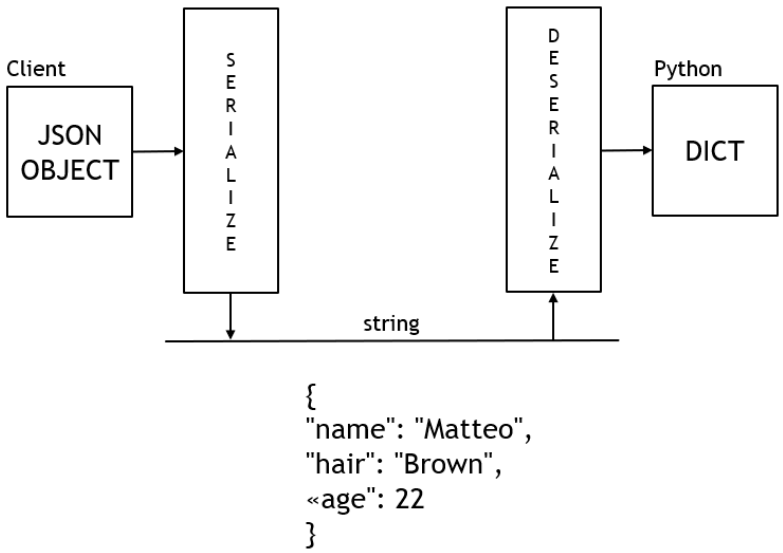

Figura 2. Trasmissione dati in formato JSON

JSON è basato su due strutture: un insieme di coppie di valori (equivalente di un dizionario Python) e un insieme ordinato di valori (equivalente di un array). Ogni linguaggio di programmazione è in grado di leggere questi tipi di strutture, rendendo JSON uno dei formati più utilizzati per lo scambio di informazioni. Inoltrando una richiesta al server e deserializzando la stringa ottenuta dalla sua risposta JSON, è quindi possibile ottenere un insieme di dati sotto forma di dizionario.

1.2 Cosa sono le API

Un'API (Application Programming Interface) definisce come accedere alle risorse e alle funzioni di un'applicazione, condividendo solo le parti necessarie, senza dare accesso all'intero codice sorgente [3].

Presenta un'interfaccia che ci permette di comunicare con il software senza sapere cosa c'è a livello più profondo. La finalità è ottenere un'astrazione migliore tra l'hardware e il programmatore o tra software a basso ed alto livello, semplificando così il lavoro di programmazione. Le API sono utilizzate per la creazione di programmi, applicazioni web, mobile, plugin e molto altro, contribuendo alla creazione di interazioni più fluide e integrate tra diversi sistemi.

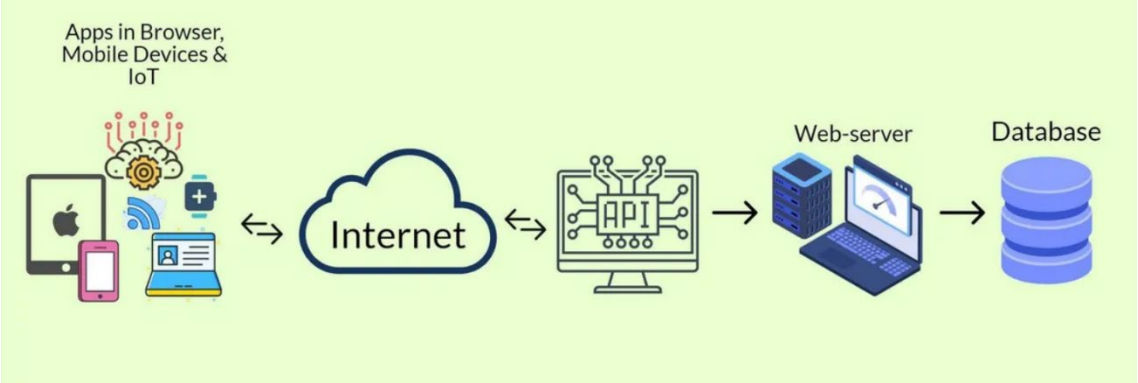

Figura 3. Schema Web API (Photo by – leadmine.net)

Esistono diverse tipologie di API. Quelle trattate in questa tesi sono le Web API, che consentono la comunicazione tra diverse Web App o tra diverse parti della stessa, o anche con un client esterno. Per lo sviluppo delle Web API viene adottato un approccio architetturale chiamato REST (Representational State Transfer), che si basa sul protocollo HTTP. Il termine REST appare per la prima volta nella tesi di dottorato dell'informatico Roy Fielding [4] ed è un approccio che tiene conto di alcune caratteristiche fondamentali:

- Risorse accessibili tramite endpoint URL
- Utilizzo del formato JSON (o XML)
- Senza stato (ogni richiesta e risposta è indipendente dalle precedenti)
- Stessi metodi di HTTP

I messaggi di richiesta e risposta di una REST API sono quindi, per convenzione, molto simili a richieste e risposte HTTP e seguono sempre il modello client-server. Spesso con il termine API si designano le librerie software di un linguaggio di programmazione, sebbene più propriamente le API sono il metodo con cui le librerie vengono usate per sopperire ad uno specifico problema di scambio di informazioni. Anche Discord ha una sua API per gli sviluppatori che sarà esaminata in seguito in questo capitolo.

1.3 Discord

Discord è una piattaforma di comunicazione vocale e testuale progettata per creare comunità online; è ampiamente utilizzato da giocatori, sviluppatori e altri utenti di vario genere. Offre una serie di caratteristiche che lo rendono unico e popolare tra gli utenti. Le sue principali funzionalità includono:

- Chat vocale e testuale: Discord permette agli utenti di comunicare in entrambi i modi, con la possibilità di partecipare a chat private o di gruppo.
- Canali e Server: le conversazioni di gruppo sono organizzate in server, a loro volta organizzati in canali. Questo consente agli utenti di organizzare la comunità al meglio.
- Integrazioni e Bot: Discord offre la possibilità di integrare diverse applicazioni e servizi, consentendo l'automazione di alcune attività e l'arricchimento delle conversazioni. I bot, ad esempio, possono essere aggiunti ai server per fornire funzionalità aggiuntive come moderazione, musica, giochi e altro ancora.

Discord ha guadagnato popolarità grazie alla sua facilità d'uso e alle numerose funzionalità offerte. È diventato un punto di riferimento per la comunicazione online, fornendo lo spazio virtuale ideale per la socializzazione, la collaborazione e lo sviluppo di comunità.

1.3.1 Discord developer portal

Discord offre un'interfaccia API, il Discord developer portal [5], che consente agli sviluppatori di interagire con la piattaforma. Questa API permette di inviare e ricevere messaggi, gestire canali, utenti e altre funzionalità offerte da Discord.

L'API di Discord si basa su due livelli principali:

- Una API HTTPS/REST per le operazioni generali. Le richieste vengono fatte a degli endpoint specifici forniti dall'API utilizzando metodi HTTP come GET e POST. Questa API REST permette agli sviluppatori di interagire con Discord in modo asincrono e di automatizzare molte funzionalità comuni.
- Una connessione basata su WebSocket per la rilevazione di eventi in tempo reale. WebSocket è un protocollo di comunicazione bidirezionale che consente una connessione persistente tra il client (il bot) e il server di Discord. Questa connessione WebSocket è utilizzata per ricevere e inviare eventi, come messaggi di chat, aggiornamenti di stato, notifiche di presenza degli utenti, modifiche ai canali e altro ancora. La connessione WebSocket è sicura e criptata, consentendo una comunicazione affidabile tra l'applicazione e Discord.

L'utilizzo combinato di questi due livelli dell'API consente agli sviluppatori di creare Bot reattivi e interattivi. Il Discord Developer Portal offre anche un'interfaccia per la gestione dei bot. Gli sviluppatori possono creare nuovi bot, impostarne le autorizzazioni, configurare i permessi, generare token di accesso e gestire eventi e interazioni del bot. È possibile anche visualizzare le statistiche di utilizzo del bot e monitorare le richieste API effettuate dal bot stesso.

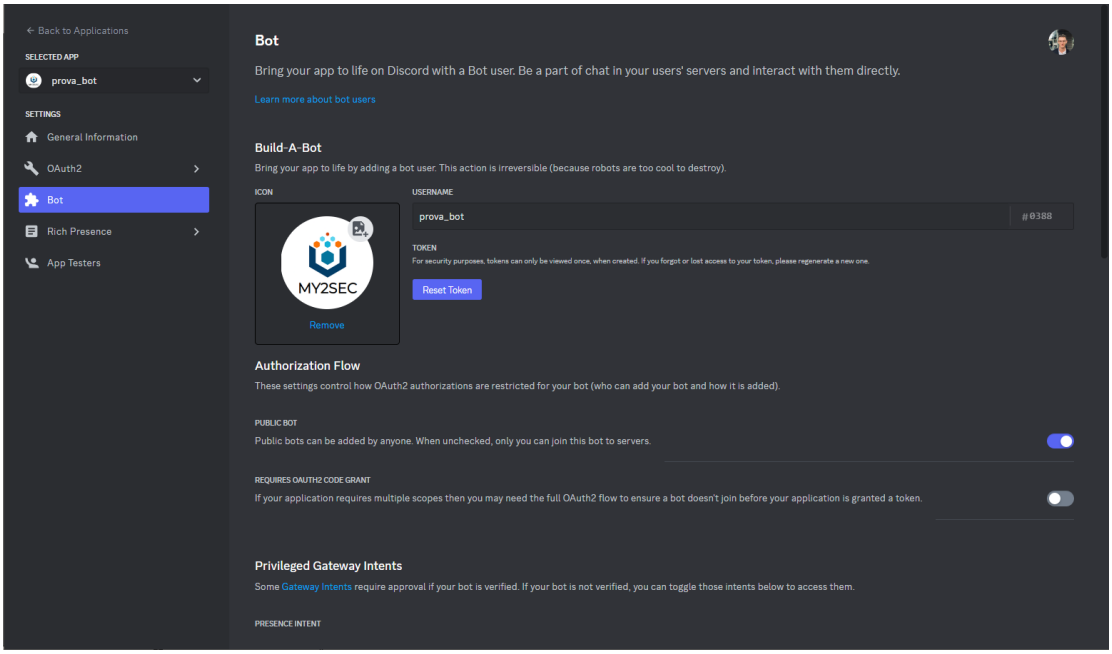

Figura 4. Discord developer interface

1.3.2 Discord Bot

I Bot sono dei piccoli programmi che girano all'interno di Discord la cui principale funzione è automatizzare dei compiti specifici. Funzionano esattamente come dei profili utente, con cui puoi dialogare e che puoi aggiungere a server o canali.

Per realizzare un Bot è necessario accedere al Discord developer portal, registrarsi e creare una nuova applicazione. Così facendo verrà generato un "Client ID" che identifica l'applicazione nel sistema di Discord [6]. Dopodiché si dovranno selezionare i permessi che si vogliono concedere al Bot. Nel nostro caso al Bot sono concessi tutti i permessi (permessi da administrator), ma è possibile rimuovere alcune autorizzazioni se desiderato.

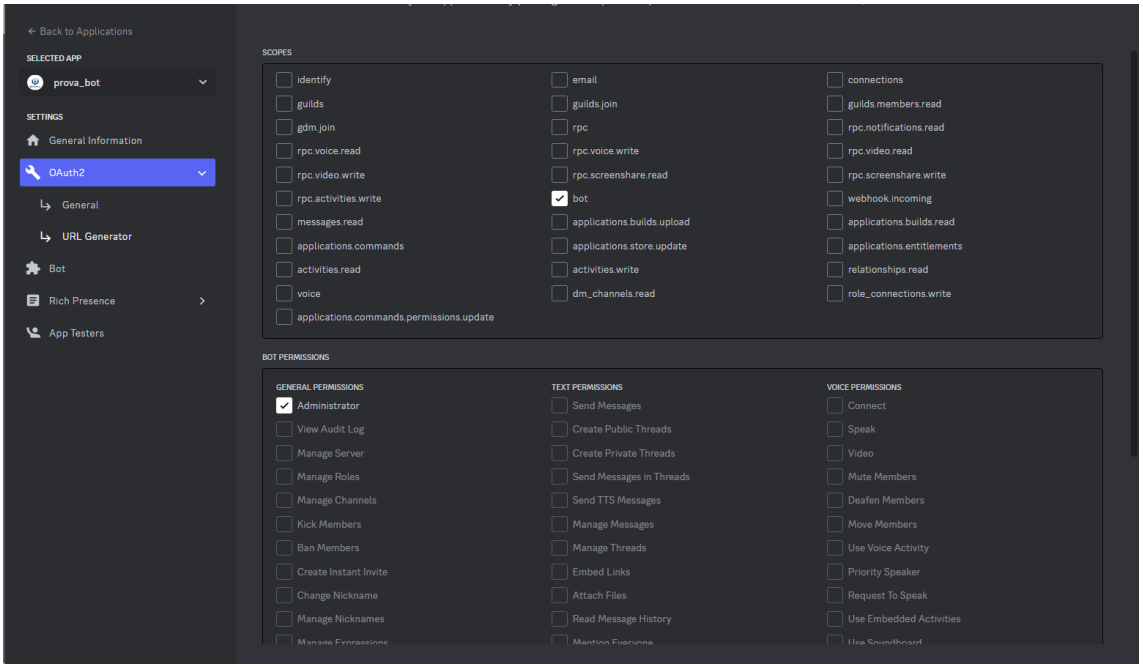

Figura 5. Permessi del Bot

Una volta configurato il Bot, occorre attivarlo e collegarlo ad un server. Questo processo avviene seguendo protocollo OAuth (Open Authorization). L'autorizzazione OAuth 2.0 (versione in uso) è un framework di autorizzazione standard utilizzato per consentire a un'applicazione di accedere a determinate risorse protette su un server o di agire a nome di un utente autenticato. Questo processo di autorizzazione è spesso utilizzato per integrare applicazioni di terze parti con piattaforme come Discord. Durante il processo di autorizzazione OAuth, viene utilizzato un callback URL. Questo è l'URL a cui il provider di autenticazione reindirizza l'utente dopo aver completato l'autorizzazione. Quando si avvia questo processo, l'utente viene quindi reindirizzato a una pagina di autorizzazione di Discord. Questa pagina chiederà all'utente di consentire all'applicazione di accedere al proprio server.

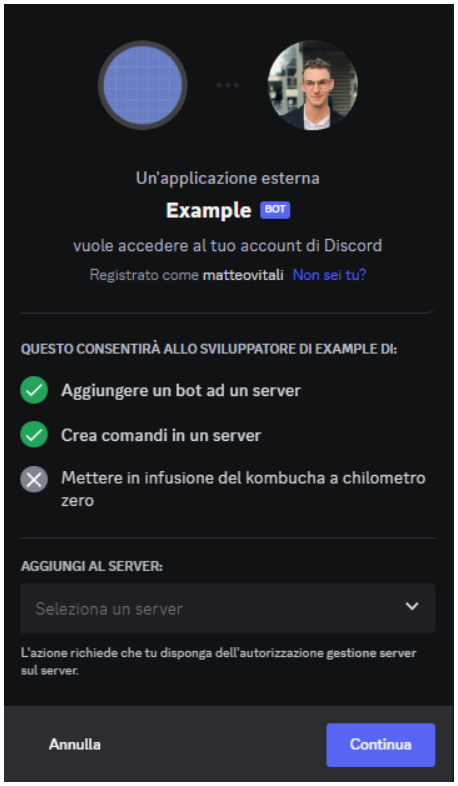

Figura 6. OAuth per connettere Bot e server Discord

Una volta fatto questo, l'applicazione può utilizzare il codice di autorizzazione ricevuto nel callback URL e il Client ID per richiedere un token di accesso al provider di autenticazione. Il token (oscurato per motivi di sicurezza) presente in figura 4 viene utilizzato per l'autenticazione durante le richieste HTTP e per stabilire la connessione WebSocket, in pratica è ciò che consente di effettuare richieste API al server Discord. È la chiave di accesso al Bot e verrà utilizzato in seguito nel codice Python per stabilire il collegamento.

L'utilizzo di bot è sempre più diffuso grazie ai loro vantaggi in termini di efficienza, automazione e miglioramento dell'esperienza utente. Consentono di ridurre il carico di lavoro manuale, fornire risposte immediate e coerenti e migliorare l'interazione tra utenti e applicazioni.

1.4 SEPA

SEPA [7] (SPARQL Event Processing Architecture) è un'architettura progettata per rilevare in modo efficiente cambiamenti di *Linked Data*. Segue un modello *publish-subscribe* che rende possibile pubblicare eventi e consentire ai nodi interessati di sottoscriversi ad essi. In questo modo si garantisce una notifica immediata dei cambiamenti o delle nuove informazioni presenti all'interno del nostro archivio. L'implementazione di SEPA si basa su tecnologie e linguaggi del Web Semantico [8], come RDF (Resource Description Framework), OWL (Web Ontology Language) e SPARQL (SPARQL Protocol and RDF Query Language) svolgendo un ruolo fondamentale come intermediario tra client e database.

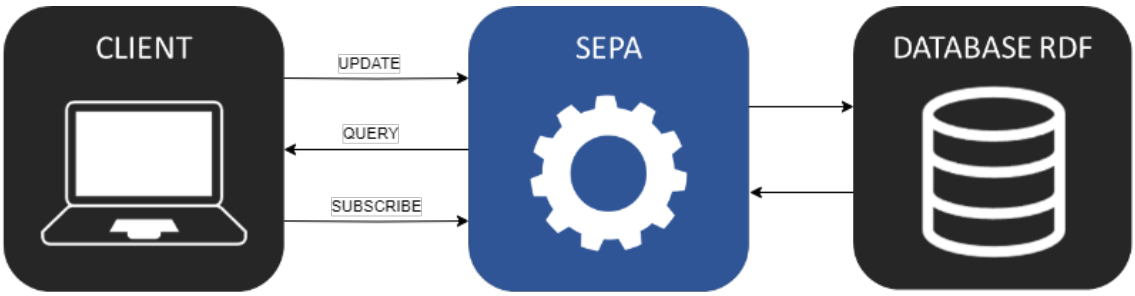

Figura 7. Relazione tra SEPA e database

Un database è una collezione strutturata di informazioni, memorizzate in modo permanente. Può essere considerato come un archivio elettronico in cui i dati possono essere memorizzati, organizzati e recuperati in modo rapido ed efficace. Per interagire con il database e manipolare i dati, viene utilizzato un linguaggio di interrogazione chiamato "query language". Questo linguaggio permette di formulare richieste per recuperare, aggiornare o eliminare dati. Come citato sopra, in questa tesi si prenderà in considerazione lo SPARQL [9] che è specificamente progettato per l'estrazione di informazioni da database semantici.

I database semantici mappano i dati seguendo il Resource Description Framework (RDF) come modello [10], il quale permette di identificare ogni risorsa tramite URI (come visto nel Capitolo 1.1 su HTTP). La componente chiave di questo tipo di struttura dati è costituita da tre elementi: soggetto, predicato e oggetto; per questo è anche detta “tripla”. Ogni informazione all’interno del database ha queste tre caratteristiche, infatti le risorse sono organizzate sotto forma di una tabella con tre colonne (soggetto, predicato, oggetto) in cui ho una tripla in ogni riga.

SOGGETTO	PREDICATO	OGGETTO
Matteo	HasHairColor	Brown
Mario	HasHairColor	Blonde
Matteo	HasAge	22
Mario	HasAge	30
Mario	Has	Car1
Car1	HasColor	Red
Car1	HasTarga	CD034PG

Figura 8. Struttura dati in un database semantico

Ovviamente i dati saranno interconnessi tra loro, quindi un oggetto di una riga può essere il soggetto di un’altra, il quale oggetto può essere soggetto di un’altra ancora e così via[1]. Una rappresentazione grafica della struttura dei dati è quella che noi chiamiamo grafo.

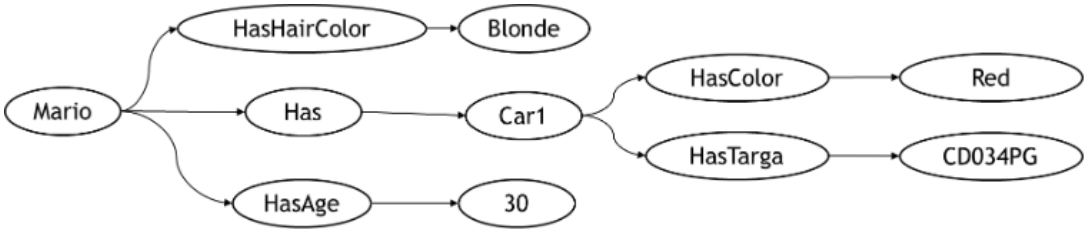

Figura 9. Grafo triple RDF

[1] Nell’esempio in figura 8, ‘Mario’ è un soggetto, non mi dice che la persona si chiama Mario. Solitamente ogni elemento da rappresentare ha un soggetto di partenza costituito da un codice identificativo univoco universale (UUID). Se volessi rappresentare una persona di nome Mario dovrei generare un UUID con gli attributi di essere una persona e di chiamarsi Mario.

Per aggiungere o ricevere informazioni è possibile ‘interrogare’ l’archivio. Le modalità con cui si comunica con il database sono tre:

- UPDATE: per aggiungere dei dati
- QUERY: per ricevere dai dati
- SUBSCRIPTION: può essere visto come caso particolare di una query. Permette di rimanere iscritti a un determinato grafo e, quando un dato cambia, ricevere in risposta i dati cambiati (aggiunti o rimossi). Questo metodo differisce dagli altri perché rende necessario lavorare in maniera asincrona ed è implementato grazie all’algoritmo del SEPA.

La risposta di una query o subscription (di tipo SELECT) è organizzata in formato JSON, sarà quindi composta da una serie di array e dizionari. Senza entrare nel dettaglio, è utile sapere che i valori dei risultati veri e propri si trovano in un sotto-array denominato ‘bindings’. Ogni binding è una coppia variabile-valore, ovvero rappresenta il valore che deve assumere una variabile in risposta ad una query. Il membro "bindings" contiene un array di elementi, uno per ogni soluzione della query. Una funzionalità specifica introdotta dal SEPA è la possibilità di passare in ingresso ad un update, query o subscription dei bindings forzati: i forcedBindings. I valori dentro i forcedBindings vanno a sostituire le variabili specificate dalla query. Questo sarà sfruttato per cambiare il grafo di sottoscrizione e per passare delle stringhe durante gli update.

A scopi di testing, utilizzeremo una dashboard dedicata di SEPA, progettata per consentire un monitoraggio intuitivo e rapido delle attività all'interno dell’archivio. La dashboard permette di visualizzare gli eventi, le query SPARQL, gli aggiornamenti e altre informazioni rilevanti riguardanti il funzionamento del database.

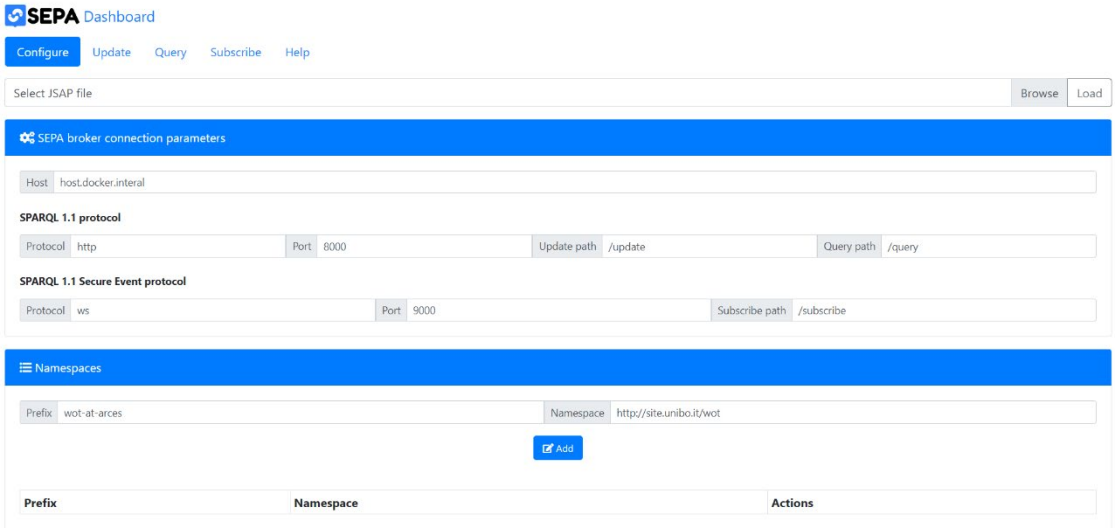

Figura 10. SEPA Dashboard

1.4.1 PAC

SEPA implementa il modello di progettazione PAC (Producer-Aggregator-Consumer) offrendo un set di componenti riutilizzabile ed estendibile. Questo modello definisce tre tipologie di componenti che possono essere utilizzate all'interno dell'architettura:

- **Produttore**: corrisponde a un editore (nel contesto di SEPA). Genera i dati e li pubblica nell'archivio utilizzando un'operazione di aggiornamento (UPDATE).
- **Consumatore**: corrisponde a un abbonato. Preleva il dato e lo invia al destinatario richiesto (nel nostro caso Discord). Il meccanismo di prelevamento può avvenire tramite interrogazione (QUERY) o sottoscrizione (SUBSCRIPTION) a seconda delle esigenze dello sviluppatore.
- **Aggregatore**: svolge entrambi i ruoli. Preleva il dato, lo elabora secondo le necessità specifiche e lo reinserisce nel database. Pertanto, l'aggregatore utilizza sia operazioni di aggiornamento che operazioni di interrogazione o sottoscrizione.

L'adozione del modello di progettazione PAC favorisce l'interoperabilità e il riutilizzo dei componenti dell'applicazione. Si consiglia agli sviluppatori di seguire questo modello per garantire una progettazione efficiente e scalabile delle applicazioni basate su SEPA.

1.4.2 JSAP

Produttori, aggregatori e consumatori, per funzionare, hanno bisogno di avere un collegamento con SEPA e devono avere accesso a tutti gli update e tulle le query. La configurazione iniziale può quindi diventare un processo lungo e ripetitivo. Per semplificare questa configurazione si utilizza un file chiamato JSAP, una versione arricchita del JSON.

Lo scopo principale del JSAP è quello di rappresentare in modo semantico cosa può fare un'applicazione SEPA. Nella tesi è stato utilizzato sia per configurare la Dashboard SEPA che per configurare consumatore e aggregatori.

La struttura del JSAP presenta una parte iniziale obbligatoria contenente le informazioni base per stabilire una connessione con il server. Questa contiene:

- HOST: il nodo di rete a cui ci si sta connettendo.
- PORTE: attraverso le quali avverrà lo scambio di informazioni.

Esempio:

```
"host": "nodo a cui connettersi",
"oauth": {
        "enable": false,
        "register": "https://localhost:8443/oauth/register",
        "tokenRequest": "https://localhost:8443/oauth/token"
},
"sparql11protocol": {
        "protocol": "http",
        "port": ‘numero della porta HTTP’,
        "query": {
                "path": "/query",
                "method": "POST",
                "format": "JSON"
        },
        "update": {
```

```
                "path": "/update",
                "method": "POST",
                "format": "JSON"
        }
},
"sparql11seprotocol": {
        "protocol": "ws",
        "availableProtocols": {
                "ws": {
                        "port": ‘numero della porta WebSocket’,
                        "path": "/subscribe"
                },
                "wss": {
                        "port": 9443,
                        "path": "/secure/subscribe"
                }
        }
},
```

- GRAFI (parte opzionale)
- NAMESPACES: sono i prefissi, utilizzati per riferirsi agli identificativi delle risorse.
- EXTENDED: contiene informazioni aggiuntive che potrebbero tornare utili allo sviluppatore. Può anche essere lasciata vuota. Nel nostro caso sono presenti TOKEN e CHANNEL_ID, parametri necessari per il collegamento a Discord.

```
"graphs": {


},
"namespaces": {
        "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
        "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
        "xsd": "http://www.w3.org/2001/XMLSchema#",
        "owl": "http://www.w3.org/2002/07/owl#",
        "time":"http://www.w3.org/2006/time#",
        "my2sec": "http://www.vaimee.it/ontology/my2sec#",
        "vaimee": "http://www.vaimee.it/ontology#",
        "discord": "http://www.vaimee.it/ontology/discord#",
        "serra": "http://www.vaimee.it/ontology/serra#",
        "error": "http://www.vaimee.it/ontology/error#"
```

```
},
"extended": {
        "discordConfig": {
                "CHANNEL_ID": ‘ID del canale’,
                "TOKEN": ‘token di autenticazione del bot’
        }
},
```

La seconda parte del JSAP contiene un elenco di tutti gli update e di tutte le query, descritte in formato JSON.

```
"updates": {
        "SEND_DISCORD_MESSAGE":{
                "sparql":"INSERT { GRAPH ?message_graph {
                                ?uuid rdf:type discord:Message;
                                my2sec:messageValue ?message_value;
                                my2sec:sourceAggregator ?source;
                                time:inXSDDateTimeStamp ?timestamp }}
                                WHERE{ BIND(UUID() AS ?uuid) BIND(now() AS
                                ?timestamp)}",
                "forcedBindings": {
                        "message_graph":{
                                "type":"uri",
                                "value":"grafo dei messaggi Discord"
                        },
                        "message_value": {
                                "type": "literal",
                                "value": "testo"
                        },
                        "source": {
                                "type": "uri",
                                "value": "http://www.vaimee.it/sources"
                        }
                }
        },
},
```

```
"queries": {
        "ALL_DISCORD_MESSAGES":{
                "sparql":"SELECT * WHERE { GRAPH ?message_graph { ?uuid
                rdf:type discord:Message; my2sec:messageValue
                ?message_value; my2sec:sourceAggregator ?source;
                time:inXSDDateTimeStamp ?timestamp }}",
                "forcedBindings": {
                        "message_graph": {
                                "type": "uri",
                                "value": "grafo dei messaggi Discord"
                        }
                }
        },
}
```

In questo esempio abbiamo un update che aggiunge un elemento al grafo dei messaggi Discord e una query che restituisce tutti gli elementi presenti in questo grafo.

Utilizzare un file JSAP non rende solo la configurazione più semplice, ma consente di tenere alcune parti fuori dal codice Python. Questo permette di modificare alcune funzionalità del Bot cambiando il JSAP, lasciando intatto il codice. Altri dettagli a riguardo saranno forniti nel capitolo successivo.

2 Sviluppo del bot in Python

Questo capitolo si concentra sull'aspetto pratico dello sviluppo software, mettendo in pratica le conoscenze teoriche presentate nel capitolo precedente. Saranno illustrate le implementazioni degli elementi teorici, consentendo ai lettori di comprendere come tradurli in soluzioni concrete.

Come visto in precedenza il Bot è composto da una serie di aggregatori e da un consumatore, comune a tutti. La logica dietro questa struttura è fare in modo che gli aggregatori mandino i dati rielaborati a un unico grafo, contenente i messaggi Discord. Il consumatore, quando viene aggiunto un nuovo dato in questo grafo, lo preleva e lo invia nel canale Discord scelto.

Un messaggio Discord ha quattro attributi ed è quindi, descritto e archiviato attraverso quattro triple:

- la prima per definire che l'elemento in questione è un messaggio Discord
- la seconda contiene il testo del messaggio (*message_value*)
- la terza contiene la sorgente (*source*)
- la quarta il tempo di invio (*timestamp*)

SOGGETTO	PREDICATO	OGGETTO
UUID	rdf:type	discord:Message
UUID	my2sec:messageValue	'testo del messaggio'
UUID	my2sec:sourceAggregator	'sorgente'
UUID	time:inXSDDateTimeStamp	'12:00'

Figura 11. Esempio di messaggio Discord archiviato nel database

Anche la dashboard di SEPA permette di visualizzare, come risposta di una query, un messaggio di Discord. Rende evidente l'UUID e i suoi quattro attributi.

uuid	type	message_value	source	timestamp
urn:uuid:186bb470-238a-4bdd-b6bf-a2c3036a16b5	http://www.vaimee.it/ontology/discord#Message	contenuto di un messaggio Discord	aggregatore sorgente	2023-06-30T13:42:36.353401210Z

Figura 12. Visualizzazione di un messaggio Discord su dashboard SEPA

Un aggregatore si occupa di riempire la seconda e la terza tripla. Fornisce quindi i valori ‘message_value’ e ‘source’, gli attributi di maggior interesse. Il valore source è semplicemente l’URI dell’aggregatore stesso. Il consumatore invia a Discord i dati di queste quattro triple (con una minima formattazione) sotto forma di stringa.

Per comprendere meglio la struttura complessiva del Bot è utile elencare le varie componenti e le varie interazioni tra di esse. Le interazioni fondamentali sono 3:

- *Subscription* dell’aggregatore al grafo di interesse (login, errori, temperature, ...). Questo consente di rilevare quando il produttore ha generato un nuovo dato.
- *Update* dell’aggregatore sul grafo dei messaggi Discord. L’aggregatore invia a questo grafo i parametri ‘message_value e ‘source’.
- *Subscription* del consumatore al grafo dei messaggi Discord. Questo consente di rilevare quando è stato aggiunto un messaggio e quindi, inviarlo.

La figura 15 presenta un esempio riguardante l’aggregatore dei login (ovvero per notificare ad un ipotetico amministratore l’accesso di utenti ad un’applicazione):

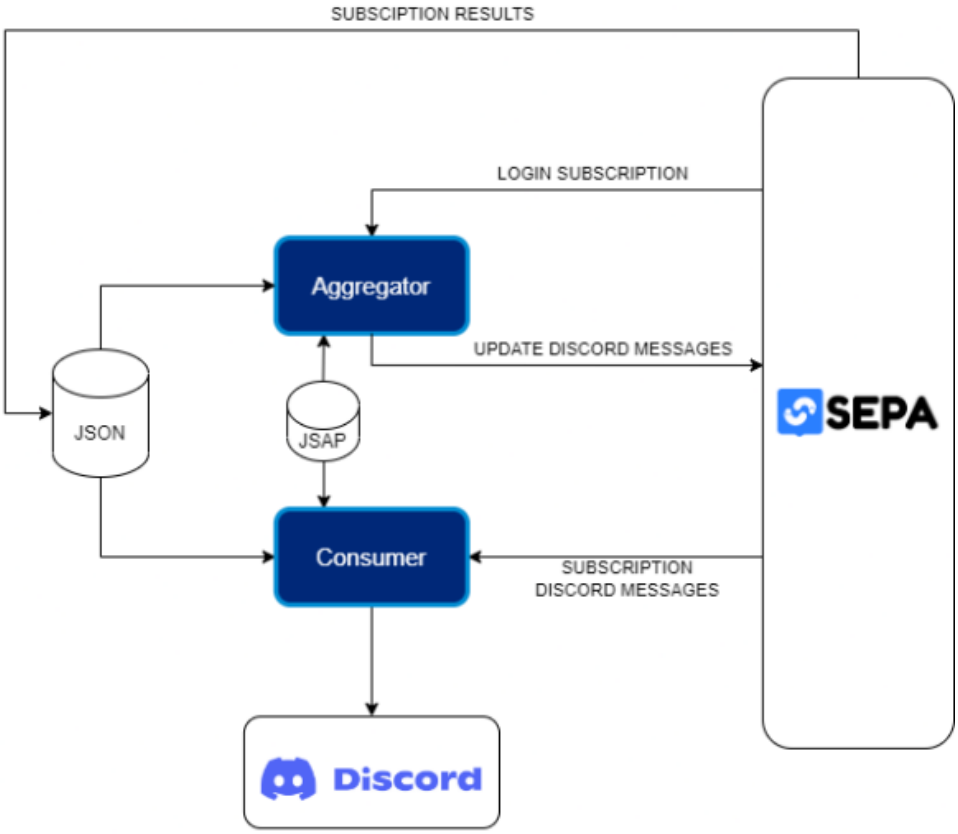

Figura 13. Schema Discord Bot

Per vedere un esempio di flusso del dato vedere il sotto-capitolo 2.3.

2.1 Librerie

Una libreria Python è un insieme di moduli e funzioni che estendono le capacità del linguaggio Python di base. Le librerie sono sviluppate da terze parti e offrono una vasta gamma di funzionalità che possono essere utilizzate dagli sviluppatori per semplificare il processo di sviluppo di applicazioni.

Esiste un indice, il Python Package Index (*PyPI*) [11] che organizza una straordinaria gamma di librerie, adatte a coprire ogni caso d'uso immaginabile. *PyPI* è un vasto repository di pacchetti Python open source offerti dalla comunità mondiale di sviluppatori.

È possibile scaricare questi pacchetti ed avere accesso alle funzionalità offerte dalla libreria. Per fare ciò esistono diversi metodi, quello consigliato [12] è l'utilizzo di *pip*, che, *de facto*, è il package manager del mondo Python. Può installare pacchetti da molte fonti, ma *PyPI* è la fonte principale per cui viene utilizzato. Durante l'installazione, pip risolverà prima le dipendenze, verificherà se sono già installate sul sistema e, in caso contrario, le installerà. Tutto ciò, di default, avviene a livello globale, installando tutto sulla macchina in un'unica posizione che varia a seconda del sistema operativo.

Per installare una libreria con pip è sufficiente digitare a terminale il comando:

`pip install 'nome libreria'`

Una volta installata, la libreria andrà importata con la direttiva:

`import 'nome della libreria'`, da inserire all'inizio del codice

In questa tesi sono utilizzate due libreria esterne: una per Discord e una per SEPA. Entrambe verranno esaminate nelle sezioni seguenti. Oltre a queste sono utilizzate anche librerie interne come *json*, *os* e *sys*.

2.1.1 Libreria Discord

Per effettuare le operazioni riguardanti Discord sarà utilizzata la libreria 'discord.py', realizzata dalla community di Discord e mantenuta da rapptz [13]. *discord.py* è un wrapper API moderno, facile da usare, e progettato per supportare operazioni asincrone. Un wrapper, in generale, è un framework o un'entità utilizzata per nascondere o incapsulare un altro oggetto. Un wrapper API è un pacchetto o una libreria specifica per un determinato linguaggio di programmazione che avvolge le complessità delle chiamate API sottostanti e fornisce metodi e funzioni più facili da utilizzare per interagire con l'API di riferimento. Nel nostro caso, ovviamente, questa libreria ci consente di comunicare con l'API di Discord.

Le caratteristiche principali di questa libreria sono:

- API Pythonic moderna che sfrutta la sintassi async/await.
- Gestione intelligente del limite di velocità che previene gli errori 429 (Too Many Requests).
- Estensione (*discord.ext.commands*) apposita per i comandi, mirata alla creazione di Bot.
- Design orientato agli oggetti che facilita l'utilizzo e l'estensione della libreria.
- Ottimizzazione per la velocità e l'efficienza della memoria.

La libreria offre due estensioni principali: *discord.ext.commands*, menzionata in precedenza, e *discord.ext.tasks*, che facilita l'impiego di task asincroni. Le estensioni non saranno utilizzare per lo sviluppo[2] ma è importante notare la loro esistenza e il valore che possono aggiungere nella creazione di applicazioni Discord.

[2] Nonostante ciò, si vedrà un esempio di utilizzo dell'estensione *discord.ext.commands* nell'ultimo capitolo sugli sviluppi futuri.

Qui di seguito sono elencate le funzionalità della libreria che verranno utilizzate nel codice:

- *class* discord.**Client**(*, *intents*, ***options*): Rappresenta una connessione al client che si connette a Discord. Questa classe viene utilizzata per interagire con l'API e il WebSocket di Discord. In ingresso necessita di un parametro intents.
- *class* discord.**Intents():** Incapsula un flag di intenti per il gateway (il Bot) di Discord. Viene utilizzato per disabilitare determinate funzionalità del gateway che non sono necessarie per eseguire il bot. Per utilizzarlo, viene passato come argomento `intents` della classe `Client` appena vista.
- **@event:** è un decoratore che registra un evento da ascoltare. Questo evento deve essere una coroutine.
- function **on_ready**(): funzione asincrona utilizzata insieme al decoratore **@event**. Chiamata quando il client ha finito di preparare i dati ricevuti da Discord.
- method **.get_channel**(*id*): metodo per oggetto Client. Restituisce un oggetto canale con l'ID specificato. L'ID, nel presente caso, è il parametro CHANNEL_ID all'interno del file JSAP.
- method **.send**(*content=None*, *, *tts=False*, *embed=None*, *embeds=None*, *file=None*,…): metodo per oggetto canale. Permette di inviare un messaggio di testo (o altro) al canale specificato.
- method **.run**(*token*, *, *reconnect=True*, *log_handler=...*,…): metodo per oggetto Client. È una chiamata bloccante che gestisce l'inizializzazione del ciclo degli eventi in automatico. Senza questa il Bot non può compiere alcuna azione. Richiede in ingresso il token di autenticazione. Questo token, nel presente caso, è il parametro TOKEN all'interno del file JSAP.

Una documentazione completa dalla libreria è disponibile sul sito ufficiale [14].

2.1.2 Libreria SEPA

Per effettuare le operazioni riguardanti SEPA (update e subscription) sarà utilizzata la libreria ‘sepy.py’, realizzata da Fabio Viola e Francesco Antoniazzi. La libreria è composta da cinque moduli che possono essere utilizzati separatamente per diversi scopi:

- SEPA: Una classe di basso livello volta a sviluppare un client per SEPA
- SAPObject: Una classe per la gestione di file JSAP
- ConnectionHandler: Una classe per la gestione delle connessioni
- Exceptions
- tablaze: Uno script eseguibile (o richiamabile come funzione) per visualizzare in modo leggibile l'output di SEPA

Per lo svolgimento del progetto saranno utilizzati solo i primi due moduli.

La classe SEPA è il costruttore ed è la responsabile del collegamento con l’*engine*. SAPObject invece è un pacchetto che serve a leggere il JSAP, consentendo di avere accesso a tutte le informazioni al suo interno. Il suo scopo è fornire al costruttore un dizionario Python creato come specificato nella documentazione di SEPA. Una volta fatto questo, è possibile utilizzare tutte le funzionalità di update, query e subscription.

Per effettuare richieste saranno utilizzati due metodi per l’oggetto Client:

- `.update(self, sapIdentifier, forcedBindings={}, host=None, token_url=None,…):` effettua un aggiornamento con il tag dell'elemento 'sapIdentifier', ovvero il nome dell’update presente nel JSAP (nel nostro caso è ‘SEND_DISCORD_MESSAGE’). È possibile fornire 'forcedBindings' come dizionario; questi conterranno il testo del messaggio Discord, la sorgente e un altro parametro *message_graph* (più avanti si descriverà la funzione di quest’ultimo). È possibile anche fornire *host*, *token_url* e *register_url* per sovrascrivere i valori SAP (se presenti).

- `.subscribe(self, sapIdentifier, alias, forcedBindings={}, handler=lambda a, r: None, host=None, token_url=None,…)`: effettua una sottoscrizione con il tag dell'elemento 'sapIdentifier', ovvero il nome della subscription (ad esempio ‘ALL_DISCORD_MESSAGES’ per iscriversi al grafo dei messaggi Discord). È possibile fornire 'forcedBindings' come dizionario, questi conterranno il parametro *message_graph*. Differisce da update e query in quanto è necessario fornire un 'alias' e un gestore (handler) da chiamare al momento della notifica, ovvero quando un dato cambia. Nel nostro caso questa funzione sarà chiamata *on_notifcation* e riceverà in input tutti i dati aggiunti.

Una documentazione completa dalla libreria è disponibile su GitHub sul profilo degli sviluppatori [15].

2.2 Sviluppo del codice

Ora, si vedrà in dettaglio il codice. Si partirà dalla configurazione iniziale per poi analizzare la struttura di aggregatore e consumatore. Insieme ai file Python contenenti questi ultimi, viene fornito anche un file JSAP di default, contenente solo la parte obbligatoria comune a tutti i JSAP, oltre a query e update.

In totale per la configurazione sono necessarie sei variabili. Di queste, le prime tre consentono il collegamento a Discord e quindi serviranno solo al consumatore:

- MESSAGE_GRAPH: definisce in quale grafo inserire/prelevare i messaggi Discord. Cambiando questo parametro è possibile avere un flusso per gli admin ed uno per i clienti (è un esempio di come è possibile modificare il funzionamento del Bot cambiando solo il file JSAP).
- CHANNEL_ID: è l’identificativo del canale nel quale si vuole inviare il messaggio.
- TOKEN: è il token di autenticazione del Bot.

- HOST_NAME: specifica l'*host.*
- HTTP_PORT: specifica la porta HTTP.
- WS_PORT: specifica la porta WebSocket.

Come specificato in precedenza per la configurazione verrà utilizzato un file JSAP, che è progettato per contenere tutti e sei i parametri. Esistono però due metodi per la configurazione tramite JSAP:

- Aprire il JSAP di default, e modificare i parametri:
  - `host`
  - `sparql11protocol.port`
  - `saprql11seprotocol.availableProtocols.ws.port`
  - `extended.discordConfig.TOKEN`
  - `extended.discordConfig.CHANNEL_ID`
  - `message_graph`[3]

```
if len(sys.argv) == 1 :
    print("#########################################################################")
    mySAP = open("./Resources/default.jsap", 'r')
    _JSAP = json.load(mySAP)
    print("- Jsap loaded, overriding configuration")
```

Figura 14. Configurazione tramite default jsap

- Specificare il percorso di un altro file JSAP da impiegare per la configurazione.

```
else:
    if sys.argv[1] == "-jsap": # OVERRIDE WITH COMMAND LINE ARGUMENT
        print("#########################################################################")
        print("Loading custom jsap from: "+sys.argv[2])
        mySAP = open(sys.argv[2], 'r')
        _JSAP = json.load(mySAP)
        print("- Custom Jsap loaded, skipping environment variables override")
        print("Host: "+str(_JSAP["host"]))
        print("#########################################################################")
```

Figura 15. Configurazione tramite custom jsap

[3] si trova tra i forcedBindings dell'update 'SEND_DISCORD_MESSAGE' e della query 'ALL_DISCORD_MESSAGES'. È necessario modificarlo in entrambi i campi

Come evidente dalle figure 14 e 15 è possibile scegliere il metodo da utilizzare inserendo argomenti nella riga di comando (*line arguments*). Se non ne sono inseriti (*len(sys.argv) == 1*) verrà caricato il JSAP di default. Se si vuole utilizzare il secondo metodo si digiti il comando:

```
python ./yourscript.py -jsap yourpath.jsap
```

Esiste anche un terzo metodo per la configurazione iniziale che consiste nell'impostare le sei variabili di configurazione come variabili d'ambiente. Durante il lancio senza *line arguments*, se viene trovata una delle sei variabili d'ambiente nel sistema, il suo valore viene sovrascritto nel file JSAP di default.

```
try:
    _JSAP["host"]=os.environ['HOST_NAME']
    print("- Env variable 'HOST_NAME' set with value: "+str(_JSAP["host"]))
except:
    print("- Env variable 'HOST_NAME' not set, using default: "+str(_JSAP["host"]))
finally:
    pass
```

Figura 16. Configurazione tramite variabili d'ambiente

Il *try/except* mostrato in figura 16 viene eseguito per tutte e sei i parametri.

Per facilitare il lancio agli sviluppatori è stata inserita una funzione *print_help()* che, qualora non siano stati inseriti *line arguments* conosciuti, stampi a video il comando da utilizzare per il secondo metodo. Stampa, inoltre, il link alla documentazione su GitHub [16].

2.2.1 Consumer

Il consumatore è il componente del Bot responsabile dell'invio dei messaggi a Discord. L'invio di questi messaggi deve avvenire appena viene aggiunto un dato nel corrispondente grafo (specificato dal parametro *message_graph*). Il consumatore dovrà quindi aprire due istanze Client: una per Discord e una per SEPA.

Innanzitutto, è necessario creare un collegamento con Discord. Saranno quindi impostate le variabili CHANNEL_ID e TOKEN, saranno dichiarati gli intenti e sarà aperto il primo Client.

```
#DISCORD CONFIGURATION
CHANNEL_ID = int(_JSAP['extended']['discordConfig']['CHANNEL_ID'])
TOKEN = _JSAP['extended']['discordConfig']['TOKEN']
intents = discord.Intents.default()
bot = discord.Client(intents=intents)
```

Figura 17. Discord configuration

La variabile *_JSAP*, mostrata anche nelle figure precedenti, non è altro che un dizionario Python, ottenuto scomponendo il file JSAP in uso. Una volta creato l'oggetto Client (*bot*), questo potrà essere utilizzato per avviare il Bot con il metodo *.run().* Dato che è una chiamata bloccante questo comando è posto nell'ultima riga del codice.

Appena il Bot è pronto, eseguirà una sottoscrizione al grafo dei messaggi Discord. Questo implica l'apertura di un altro Client verso SEPA.

```
@bot.event
async def on_ready():
    client = SEPA(sapObject=SAPObject(_JSAP))
    client.subscribe('ALL_DISCORD_MESSAGES', 'PROVA', on_notification)
```

Figura 18. Sottoscrizione al grafo dei messaggi Discord

Come discusso nel sotto-capitolo riguardante la libreria sepy.py, il metodo subscribe necessita in ingresso di una funzione (*on_notification*), la quale viene chiamata ogni volta che un dato cambia. Durante la prima esecuzione però la sottoscrizione restituisce tutti i dati presenti nel grafo, ovvero tutti i messaggi Discord. Per sopperire a questo problema è stata inserita una variabile globale *first_results* che, di default è impostata a 1 e, dopo la prima esecuzione, viene impostata a 0. Questa variabile verrà utilizzata come condizione nella funzione *on_notification*.

```python
def on_notification(a,r):
    global first_results
    if first_results == 1:
        print("Ignored first results")
        first_results=0
    else:
        if discord_message_data(a) == []:
            return
        else:
            canale = bot.get_channel(CHANNEL_ID)
            bot.loop.create_task(canale.send(discord_message_data(a)))
            print('message sent successfully')
```

Figura 19. Funzione 'on_notification' consumatore

La funzione *on_notification* riceve in ingresso sia i dati aggiunti (*a*) che i dati rimossi (*r*). Tuttavia, i dati rimossi non sono rilevanti per lo scopo del Bot. Inoltre, se ci troviamo nella fase di prima esecuzione (*first_results == 1*) i dati ricevuti vengono ignorati.

Qualora sia stato aggiunto un nuovo dato invece, la funzione *on_notification*, provvede ad inoltrarlo a Discord. Il dato riceve una breve formattazione dalla funzione *discord_message_data()*. Dato che la libreria *discord.py* lavora in asincrono l'operazione di invio del messaggio è stata incapsulata dento un task.

2.2.2 Aggregator

L'aggregatore è il componente del Bot responsabile della formattazione del dato. La sua struttura è simile a quella del consumatore con la differenza che, una volta estratto, il dato non è inoltrato a Discord, ma viene reinserito nel database tramite un aggiornamento. Non sarà quindi presente nessuna funzionalità della libreria *discord.py*.

Anche per l'aggregatore sarà necessario aprire un Client verso SEPA, che permetterà di eseguire le operazioni di sottoscrizione e update. Il meccanismo di sottoscrizione è identico a quello del consumatore ma il grafo è diverso per ogni aggregatore. Ad esempio, l'aggregatore dei messaggi d'errore è iscritto al grafo *http://www.vaimee.it/ontology/error#messages* tramite la query 'ALL_ERROR' (figura 20).

```
#Called when a new error si received
def on_notification(a,r):
    global first_results
    if first_results == 1:
        print('first results ignored')
        first_results = 0
    else:
        client.update('SEND_DISCORD_MESSAGE', forcedBindings={
        "message_value" : error_format(a),
        "source" : "http://www.vaimee.it/sources/error_messages_aggregator"
        })

client = SEPA(sapObject=SAPObject(_JSAP))
client.subscribe('ALL_ERROR', 'PROVA', {}, on_notification)
```

Figura 20. Funzione on_notification aggregatore

Come per il consumatore, la funzione *on_notification* riceve sia i dati aggiunti che i dati rimossi e ignora i primi risultati. Quando un nuovo dato è stato aggiunto, viene eseguito un update sul grafo dei messaggi Discord ('SEND_DISCORD_MESSAGE'). Il parametro *message_value*, ovvero il testo del messaggio, viene fornito tramite forcedBindings ed è formattato da un'apposita funzione che restituisce una stringa contenente tutte le informazioni necessarie. Questa funzione è diversa per ogni aggregatore, nella seguente figura è illustrato un esempio sulla formattazione dei messaggi d'errore.

```
def error_format(res):
    if res == []:
        print('NON CI SONO ERRORI')
        return
    SOURCE = res[0]['Source']['value']
    TIME = res[0]['Time']['value'].split('T')[1].split('.')[0]
    ERRORTYPE = res[0]['ErrorType']['value']
    DESCRIPTION = res[0]['Value']['value']
    COMMENT = res[0]['Comment']['value']
    return f"Error generated from {SOURCE} AT {TIME}\\nErrorType: {ERRORTYPE}\\nComment:\\n{COMMENT}\\n-----------\\nDescription:\\n{DESCRIPTION}"
```

Figura 21. Formattazione messaggi d'errore

2.3 Esempio di funzionamento

In questo sotto-capitolo, non verranno introdotti nuovi concetti, ma verrà fornito un esempio del flusso dati del Bot. L'obiettivo è riepilogare i concetti esaminati finora e rafforzare la coerenza logica che li collega.

Prima di ciò è bene ricordare che al suo interno il Bot non ha produttori, ma solo consumatore e aggregatori. Il suo compito è rilevare l'arrivo di un nuovo dato, non verrà considerato come il dato è stato generato.

Sebbene le applicazioni siano molteplici si vedrà un esempio relativo alla notifica di nuovi login.

Aggregatore e consumatore lavorano in parallelo. Al momento dell'avvio l'aggregatore esegue una sottoscrizione al grafo degli utenti, il consumatore esegue una sottoscrizione al grafo dei messaggi Discord. Entrambi sono in attesa dell'arrivo di nuovi dati.

A questo punto il flusso può essere riassunto nei seguenti passaggi:

1) Invio di un nuovo dato (da un produttore esterno) nel grafo degli utenti. Questo in un caso di applicazione reale avviene quando un nuovo utente si iscrive alla piattaforma di interesse. Un utente ha due attributi: mail e username.

Mail	Username
http://www.vaimee.it/my2sec/Mario@vaimee.it	Mario

Figura 22. Esempio di un nuovo utente su dashboard SEPA

2) L'aggregatore rileva l'arrivo di un nuovo dato e viene chiamata la relativa funzione *on_notification*. Questa, a sua volta, chiama la funzione di formattazione, che trasforma il dato in una stringa.

```
return f"A new user has registered\\nMail: {MAIL}\\nUsername: {USERNAME}"
```

Figura 23. Esempio di formattazione dei dati di un nuovo utente

3) L’aggregatore esegue un update sul grafo dei messaggi Discord, in cui inserisce come *message_value* la stringa appena formattata. Inserisce anche l’URI dell’aggregatore come sorgente.

uuid	message_value	source	timestamp
urn:uuid:d430fc30-c99c-405a-8f30-db0bb8cf45dc	A new user has registered Mail: Mario@vaimee.it Username: Mario	http://www.vaimee.it/sources/login_aggregator	2023-07-04T14:50:15.755327734Z

Figura 24. Esempio di un messaggio Discord su dashboard SEPA

4) Il consumatore rileva l’arrivo di un nuovo dato e viene chiamata la relativa funzione *on_notification*. Questa invierà a Discord un messaggio contenente *message_value*, *source* e *timestamp*.

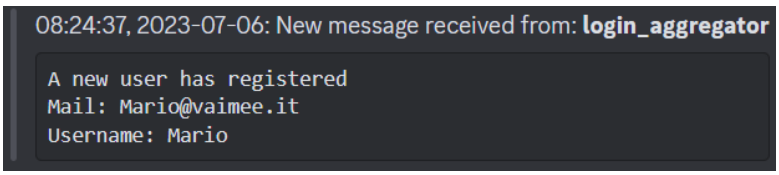

Figura 25. Notifica di un nuovo utente su Discord

3 Conclusione e sviluppi futuri

La presente tesi si poneva, come obiettivo primario, la creazione di un metodo efficace e versatile per tracciare il flusso di dati all'interno di un database semantico. Attraverso l'analisi approfondita dei concetti di base, lo sviluppo di un Bot utilizzando le tecnologie appropriate, e la dimostrazione pratica del suo funzionamento, possiamo affermare che l'obiettivo è stato pienamente raggiunto.

Il Bot sviluppato utilizza un'architettura basata su SEPA (SPARQL Event Processing Architecture) che sfrutta il paradigma *publish-subscribe* per la rilevazione e la notifica di cambiamenti nei dati dell'archivio di interesse. Attraverso l'implementazione di consumatore e aggregatori è stato possibile gestire in modo accurato e personalizzabile il flusso dei dati.

La tesi ha dimostrato quindi, come l'utilizzo di SEPA, insieme a librerie Python e l'API di Discord, possa fornire soluzioni innovative per gestire e analizzare in modo intelligente i dati in contesti complessi.

Grazie alla sua architettura modulare e scalabile, il Bot può essere facilmente adattato ed esteso per supportare nuove funzionalità e requisiti specifici.

La tesi è stata svolta presso VAIMEE srl, startup innovativa e spinoff dell'Università di Bologna, ospitata presso il CesenaLab. Quanto mostrato è stato applicato a due casi d'uso attualmente sviluppati dalla startup. Il primo è un'applicazione per la gestione dei progetti in un contesto di smart working, mentre la seconda, in campo agritech, ha come obiettivo quello di fornire una previsione sul fabbisogno irriguo delle colture.

Allo scopo di creare dei dimostratori in questi due contesti, durante la fase di stesura della tesi, alcune funzionalità sono già state aggiunte. Oltre ad una serie di nuovi aggregatori, dal funzionamento analogo a quelli già trattati, sono stati sviluppati nuovi componenti come:

- **Bot giornaliero** [17] (*Daily_Bot*): è a tutti gli effetti un produttore. Permette di impostare task giornalieri che verranno eseguiti ad un determinato orario. L'esecuzione del task produce il dato e lo carica sul grafo dei massaggi Discord tramite un update. Questo sarà poi inviato su Discord dal consumatore come per i dati ricevuti dagli aggregatori. Per lo sviluppo è stata utilizzata un'ulteriore libreria esterna: *APScheduler* [18], disponibile su *PyPI*.

  Il Bot giornaliero può essere utilizzato per inviare promemoria o riassunti delle attività a fine giornata essendo utile sia a sviluppatori che ad utenti finali.

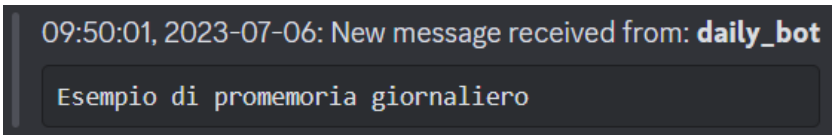

Figura 26. Promemoria ricevuto dal Daily Bot

- **Comandi** [19] (*Criteria_commands*): permettono di interagire con il database direttamente dalla chat di Discord. Quando un comando viene digitato viene eseguita una query sul database, i dati ottenuti vengono rielaborati e mandati come risposta. In questo caso, i dati non passano da un aggregatore e vengono mandati come 'embed', una struttura dati utilizzata per visualizzare contenuti avanzati all'interno dei messaggi di Discord. La scelta è puramente estetica. Un comando può prendere in ingresso dati inseriti dall'utente direttamente dalla chat permettendo di eseguire query personalizzate in modo molto semplice. Per lo sviluppo è stata utilizzata l'estensione *discord.ext.commands* della libreria *discord.py*.

  Nella figura seguente (figura 27) è mostrato come il comando */availabelwater*, specificando il parametro *feature* (si riferisce ad un terreno), permetta di visualizzare l'acqua disponibile nel terreno selezionato (*Schoonoord_1682603871847*).

Figura 27. Esempio di risposta ad un comando

Queste sono solo alcune delle funzionalità che possono essere aggiunte al Bot. Grazie alla sua struttura, pensata per favorire la scalabilità, è infatti possibile ampliarlo ulteriormente aggiungendo funzioni come la sottoscrizione personalizzata per ciascun utente, modalità avanzate per gli sviluppatori, integrazione con diverse piattaforme e molto altro ancora. L'obbiettivo finale è ottenere una fitta rete di connessioni che permetta di trattare il dato nella maniera più completa possibile al fine di poterlo utilizzare in modo strategico per prendere decisioni informate e raggiungere risultati ottimali.

Un esempio puramente teorico di come potrebbe essere ampliata questa rete è fornito in figura 28.

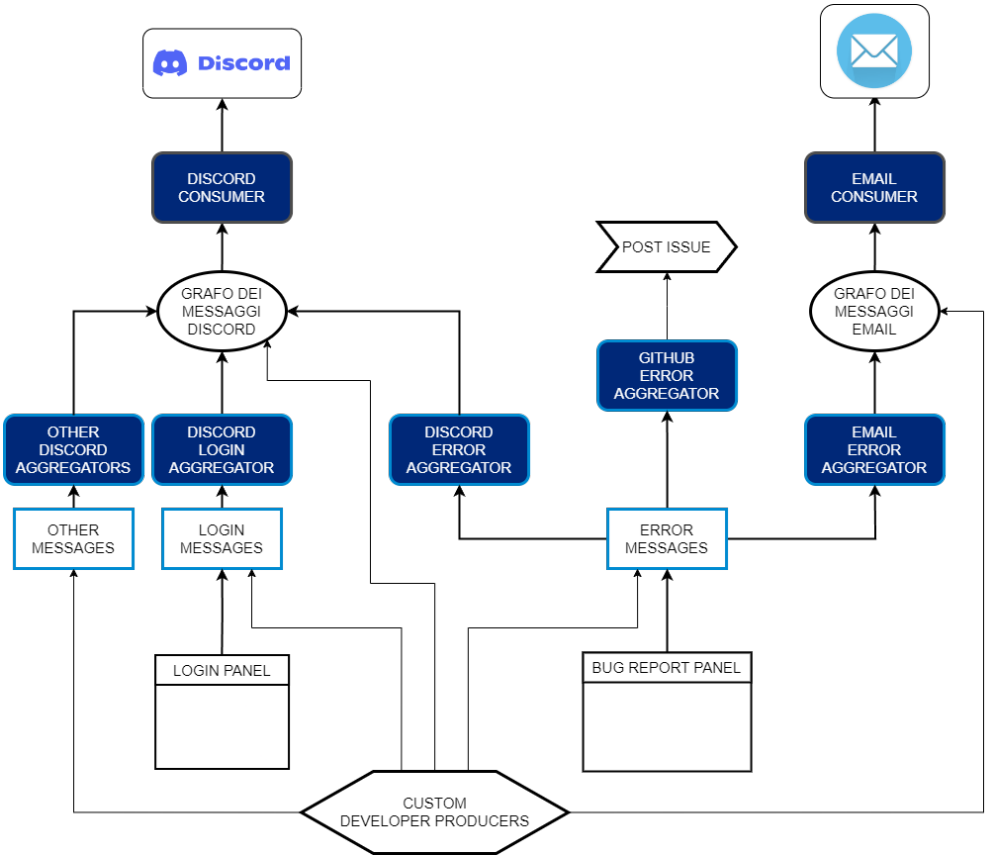

Figura 28. Ecosistema Bot (sviluppi futuri)

Bibliografia e sitografia

[1] H. S. Oluwatosin, «Client-Server Model,» *IOSR Journal of Computer Engineering (IOSR-JCE),* vol. 16, pp. 67-71, 2014.

[2] F. e. a. Pezoa, «Foundations of JSON schema.,» in *Proceedings of the 25th international conference on World Wide*, 2016.

[3] S. P. Ong, «The Materials Application Programming Interface (API): A simple, flexible and efficient API for materials data based on REpresentational State Transfer (REST) principles,» *Computational Materials Science,* vol. 97, pp. 209-215, 2015.

[4] R. T. Fielding, «Fielding Dissertation: CHAPTER 5,» 2000. [Online]. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[5] Discord, «Discord Developer Portal,» [Online]. Available: https://discord.com/developers/docs/reference.

[6] Discord, «Discord Developer Portal,» [Online]. Available: https://discord.com/developers/applications.

[7] L. Roffia, «Dynamic Linked Data: A SPARQL Event Processing Architecture,» *MDPI,* 2018.

[8] T. Berners-Lee, «The semantic web,» *JSTOR,* vol. 284, n. 5, pp. 34-43, 2001.

[9] L. Roffia, «A Semantic Publish-Subscribe Architecture for the Internet of Things,» *IEEE Internet of Things Journal,* vol. 3, n. 6, pp. 1274-1296, 2016.

[10] E. Miller, «An introduction to the resource description framework,» *ディジタル図書館,* n. 13, pp. 3-11, 1998.

[11] P. community, «PyPI . The Python Package Index,» [Online]. Available: https://pypi.org/.

[12] L. K. Kollár, «Managing Python packages the right way,» 18 Agosto 2020. [Online]. Available: https://opensource.com/article/19/4/managing-python-packages.

[13] Rapptz, «Welcome to discord.py,» 2015. [Online]. Available: https://discordpy.readthedocs.io/en/latest/index.html.

[14] D. community, «API reference,» [Online]. Available: https://discordpy.readthedocs.io/en/latest/api.html.

[15] F. Antoniazzi, «fr4ncidir/SEPA-python3-APIs: Client-side libraries for the SEPA platform (Python3),» 2020. [Online]. Available: https://github.com/fr4ncidir/SEPA-python3-APIs.

[16] M. Vitali, «Discord-bot/README.md at master · Whitefield64/Discord-bot</title,» 2023. [Online]. Available: https://github.com/Whitefield64/Discord-bot/blob/master/README.md.

[17] M. Vitali, «Whitefield64/Discord-bot at Daily-Bot,» 2023. [Online]. Available: https://github.com/Whitefield64/Discord-bot/blob/Daily-Bot/src/daily_bot.py.

[18] A. Grönholm, «APScheduler · PyPI,» 2023. [Online]. Available: https://pypi.org/project/APScheduler/.

[19] M. Vitali, «Discord-bot/src/Criteria_commands.py at master · Whitefield64/Discord-bot<,» 2023. [Online]. Available: https://github.com/Whitefield64/Discord-bot/blob/master/src/Criteria_commands.py.