Politecnico di Torino

Politecnico di Torino

Dipartimento di Automatica e Informatica

Corso di Laurea Magistrale in
Ingegneria Informatica

Orientamento Cybersecurity

ESTENDIBILITÀ E TESTABILITÀ DI UNA PIATTAFORMA PER LA GESTIONE DI UNA SUPPLY CHAIN

Relatore:
Prof. Giovanni MALNATI

Correlatore:
Prof. Fabio FORNO

Tesi di laurea di:
Riccardo TEDESCO
Matr. 269094

Anno Accademico 2021 - 2022

Ai miei genitori, a mia sorella e al team Rattlesmake.

"Se puoi sognarlo, allora esiste un modo per realizzarlo. Ma se non esiste, e ci credi davvero, rimboccati le maniche e trovalo."

[Walt Disney]

Abstract

Una *catena di distribuzione* (*supply chain*) è quel processo che permette di portare sul mercato un prodotto o un servizio. Tale processo, che coinvolge una moltitudine di persone, attività, informazioni e risorse, costituisce un'attività molto complessa che richiede, fra le altre cose, servizi di *trasparenza* e *tracciabilità* a livello di filiera produttiva.

Con tali finalità è stata, quindi, realizzata *iChain*, una piattaforma basata sugli standard *GS1*, in particolare lo standard *EPCIS* che consente di rappresentare qualsiasi evento riguardante un *flusso produttivo*.

Lo scopo di questa tesi è quello di *estendere* e *testare* la piattaforma iChain, una piattaforma che funziona in streaming, che ha uno stato complesso e distribuito e che utilizza diversi *stream processors* per effettuare analisi ed elaborazioni sugli eventi EPCIS. È stato perciò realizzato un *ambiente di test* in cui poter eseguire, anche in parallelo, una serie di test volti a verificare l'effettivo funzionamento della piattaforma, consentendo così di generare eventi EPCIS di prova o di replicare insiemi selezionati di eventi EPCIS, il tutto in modo isolato e temporaneo al fine di non corrompere, in alcun modo, né i dati non di prova memorizzati sui database della piattaforma medesima né lo stato degli stream processors. A tale scopo, l'architettura iniziale di iChain - costituita da *MongoDB*, *Kafka* e *Faust* (una libreria python per lo stream processing) - è stata integrata con un nuovo elemento: *Debezium*, grazie al quale è stato possibile ottenere un ambiente di test in cui eseguire in modo *idempotente* dei test, ciascuno dei quali contraddistinto da un particolare *prefisso* allo scopo di identificare univocamente la *configurazione di test* da usare (i.e. i nomi delle collezioni MongoDB, dei kafka topics e dei connectors).

Avere un ambiente di test con le caratteristiche di cui sopra è importante al fine di poter introdurre, in modo *incrementale*, nuove tipologie di eventi ed estensioni. In questa tesi, sfruttando l'ambiente di test realizzato, si è infatti provveduto anche a migliorare la piattaforma iChain affinché fosse in grado di gestire sia un nuovo tipo

di evento EPCIS, l'*Association Event* (che è stato definito a partire dalla versione 2.0 dello standard EPCIS), sia le *estensioni di eventi EPCIS* che permettono di rappresentare, in un evento EPCIS, anche degli attributi non direttamente codificati dallo standard EPCIS ma comunque necessari per un dato contesto di business.
Come ultima cosa, in questa tesi è stato migliorato il meccanismo, fornito da iChain, delle *regole di inoltro*, attraverso cui le aziende registrate sulla piattaforma possono specificare quali dati trasmettere a terzi e a chi. Al meccanismo preesistente sono state apportate due importanti modifiche: come prima cosa, è stato introdotto il concetto di *richiesta di collaborazione* (*partnership request*) che impedisce a un'azienda di inoltrare i propri dati a terzi, sempre mediante regole di inoltro, senza che sia stata prima stabilita una reale collaborazione tra le parti interessate; come seconda cosa, è stata creata la figura di utente con privilegi di *super amministratore*, vale a dire un utente autorizzato a leggere i dati memorizzati sulla piattaforma senza la necessità che gli vengano inoltrati espressamente, ottenendo in tal modo un significativo risparmio di spazio sui database. Tale utente aiuta anche ad automatizzare la gestione dei test.

Indice

1

Introduzione

1.1
Supply chain

Ogni giorno vengono acquistati continuamente nuovi prodotti, dai più banali, come una confezione di latte, ai più complessi, come per esempio automobili dotate di sistemi tecnologici e informatici sempre più articolati e sofisticati; allo stesso tempo, ogni giorno si usufruisce di sempre più servizi. Tuttavia, raramente ci rendiamo conto di quante persone, attività, informazioni e risorse siano coinvolte nel processo atto a fornire un prodotto sul mercato o un servizio. Tutto ciò costituisce una *supply chain* (o catena di approvvigionamento).
Il processo che porta un prodotto sul mercato è molto complesso e articolato e può essere scomposto nelle seguenti fasi (anelli) [1]:

1. approvvigionamento, cioè la fase in cui si acquisiscono le materie prime;
2. produzione, cioè la fase in cui si trasformano le materie prime in un prodotto finito;
3. trasporto, cioè la fase in cui si portano i prodotti in un deposito;
4. stoccaggio, cioè le varie fasi a cui è soggetto un prodotto in magazzino. Esse sono:
    - gestione degli ordini;
    - ricevimento del prodotto;
    - spedizione del prodotto;
    - gestione delle scorte;
5. consegna, cioè la fase in cui il prodotto viene consegnato al cliente finale, sia esso un privato cittadino o un'azienda.

A questa complessità intrinseca si aggiungono le difficoltà derivanti da leggi nazionali e comunitarie che obbligano le aziende ad avere dei sistemi di *tracciabilità* dei propri prodotti. A oggi sono pochi i sistemi di tracciabilità veramente potenti, anche perché questa operazione è spesso vista dall'azienda più che altro come un onere imposto dalle normative di settore. Al contrario, un sistema di tracciabilità forte rappresenta senza dubbio un grosso beneficio per una azienda, con diversi vantaggi quali per esempio:

1. maggiore fiducia da parte dei consumatori. Questi, infatti, possono accedere a più informazioni sulla storia di un prodotto e avere così una maggiore consapevolezza di ciò che stanno effettivamente acquistando; anche in relazione ai vari aspetti etici e sociali spesso legati a quel particolare prodotto;

2. efficienza e sicurezza. Avere un robusto sistema di tracciabilità consentirebbe a un'azienda di identificare tutti i prodotti che potenzialmente abbiano subito danni di qualsiasi natura; diversamente, l'azienda si troverebbe costretta a ritirare dal mercato, con un conseguente danno economico, una determinata quantità di merce qualora si verificasse un qualche evento dannoso come, per esempio, un lotto di prodotti alimentari andati a male o di pezzi di ricambio difettosi;

3. ottimizzazione dei processi aziendali interni. Un sistema di tracciabilità consente infatti di condurre analisi mirate sulle aree aziendali più problematiche, consentendo così di intervenire per migliorarle.

Ovviamente, per avere un valido sistema di tracciabilità, è necessario che ci sia anche la giusta comunicazione con gli stakeholders: tale comunicazione è nota come *trasparenza* della supply chain. Naturalmente non bisogna dimenticare che le aziende sono spesso in competizione tra di loro e quindi sarebbe inverosimile aspettarsi una trasparenza assoluta, perché comunque esistono informazioni e fasi del processo produttivo di un'azienda che costituiscono segreto aziendale. Tuttavia, avere un sistema che consenta di condividere il maggior numero di informazioni non sensibili fornirebbe ulteriori vantaggi alle aziende, come per esempio:

1. migliori relazioni commerciali. Un'azienda potrebbe identificare più facilmente eventuali fornitori poco affidabili e sostituirli con altri che offrono migliori garanzie;

2. ottimizzazione su larga scala dei processi aziendali. Analizzando le parti della catena di produzione non direttamente legate ai propri flussi produttivi interni, è possibile per esempio identificare i clienti in senso lato che richiedono un determinato prodotto, realizzato da terzi ma a partire da prodotti interni, consentendo all'azienda di orientare al meglio la propria produzione, operando in modo da migliorarsi relativamente agli indici di qualità che sono importanti per le filiere in cui c'è una grande domanda della sua merce e abbandonando o limitando la produzione per quelle catene produttive su cui ha una inferiore richiesta.

1.2
iChain

iChain, originariamente nata con il nome di *XTap*, è una piattaforma che offre servizi di trasparenza, tracciabilità e condivisione dei dati a livello di supply chain, sfruttando gli standard GS1 tra cui principalmente lo standard EPCIS per la rappresentazione degli eventi che si verificano in qualsiasi attività produttiva [2].
Come mostrato nella figura 1.1, iChain vuole quindi puntare a fornire una visione panoramica dell'intera filiera produttiva, non limitandosi a una visione parziale, cioè limitata a quanto avviene nella propria azienda.

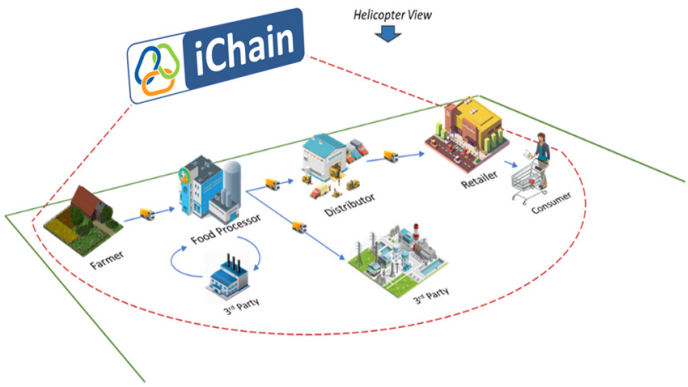

Figura 1.1: una visione di insieme della supply chain [3].

Essendo basato su EPCIS, iChain modella ogni evento usando il metodo delle *4W*: what, when, where, why.
Oltre ad essere in grado di acquisire gli eventi EPCIS, iChain fornisce una serie di tool grafici per:

- visualizzare i dettagli di un evento;
- avere una vista aggregata sui prodotti coinvolti negli eventi (what);
- avere una vista aggregata sulla tipologia di evento (why);
- avere una vista aggregata sui luoghi in cui sono avvenuti gli eventi (where);
- avere una vista aggregata temporalmente sugli eventi (when);
- visualizzare la storia/evoluzione di un prodotto;
- visualizzare il flusso produttivo di filiera;
- definire regole di trasparenza e autorizzazione, cioè definire regole che permettono di indicare cosa condividere del proprio flusso produttivo a terzi e con chi condividerlo, garantendo così a un'azienda di poter preservare le proprie esigenze di riservatezza su alcuni dati che essa gestisce. Naturalmente, tali regole non sono immutabili ma possono essere modificate in funzione di varie esigenze commerciali.

2

Gli standard GS1

Nel capitolo introduttivo, si è detto come la piattaforma iChain sia basata sugli standard *GS1*. In questo capitolo, si approfondiranno le codifiche GS1 gestite e usate da iChain; in particolare verrà trattato lo standard *EPCIS*, che costituisce di fatto il nucleo su cui si basa iChain.

2.1
Global Standards 1

GS1 (Global Standards 1) è una organizzazione non-profit che sviluppa una serie di standard volti a consentire la comunicazione intra e interaziendale. Il più celebre standard GS1 per riconoscere e identificare univocamente prodotti, documenti, pallet, persone con incarichi giuridici e amministrativi e luoghi è senza ombra di dubbio il codice a barre, ma vi è un'ampia gamma di standard aventi scopi analoghi.
Ogni codice GS1 termina con una *cifra di controllo*, o *check digit*, calcolata a partire dalle cifre precedenti in base all'algoritmo ufficiale GS1 [4].

2.2
GLN

Il *GLN* (Global Location Number) è la chiave GS1 usata per identificare sedi legali e operative, reparti interni, magazzini, CeDi, fabbriche, punti vendita, enti pubblici, banche e operatori del contante [5]. Insomma, è usato per identificare univocamente le aziende o i luoghi collegati a esse.
Un codice GLN ha una lunghezza di 13 cifre ed è così strutturato:

- un prefisso aziendale GS1 di 9 cifre. Tale prefisso è assegnato a ciascuna azienda nel momento in cui essa aderisce al sistema GS1. Qualora ne abbia la necessità, un'azienda può richiedere più di un prefisso aziendale;

- un codice di luogo di 3 cifre. Tale codice è assegnato dall'azienda stessa a ciascun luogo del proprio dominio che deve essere identificato;
- la cifra di controllo.

Se un'azienda non fa parte del sistema GS1 ma necessita per qualche ragione di un codice GLN, può comunque noleggiare un singolo GLN.

2.2.1 SGLN

Il GLN identifica, come visto, un luogo. Qualora si abbia la necessità di identificare univocamente un punto specifico all'interno del luogo è necessario usare un codice *SGLN* (Serialized GLN) che si ottiene aggiungendo in coda al codice GLN un seriale, lungo da 1 a 20 cifre, scelto dall'azienda (solitamente in maniera progressiva).
La codifica SGLN ricopre un ruolo di rilievo nello standard EPCIS, al fine di identificare univocamente un *read point* o una *business location* (il significato di tali termini sarà chiarito più avanti).

2.3 GTIN

Il *GTIN* (Global Trade Item Number) è la chiave GS1 usata per identificare i prodotti (unità consumatore) e i colli (unità imballo) in tutto il globo [6].
Esistono varie classi di codifiche GTIN, ciascuna volta a identificare oggetti diversi;

- prodotti a peso fisso: sono identificabili da GTIN-12 (il GTIN-12 è una codifica usata solo in America), GTIN-13, GTIN-8, SGTIN;
- prodotti a peso variabile: sono identificabili da GTIN-12 e GTIN-13;
- colli a peso fisso: sono identificabili da GTIN-12, GTIN-13, GTIN-14, SGTIN;
- colli a peso variabile: sono identificabili da GTIN-14.

Il numero che segue una codifica GTIN indica il numero di cifre che compongono il codice stesso (e.g. GTIN-13 ha una lunghezza di 13 cifre). Le codifiche hanno una struttura pressapoco simile che, per GTIN-13, consiste in:

- un prefisso aziendale, lungo 9 cifre;
- un codice di classe di un prodotto, lungo 3 cifre;
- la cifra di controllo.

Costituisce un'eccezione il GTIN-8, il quale è in genere usato per quei prodotti in cui lo spazio sull'etichetta o sulla confezione è minimo; per tale ragione esso ha solo un codice prodotto (7 cifre) e la cifra di controllo. Il GTIN-14, invece, oltre alla struttura suddetta presenta un altro componente, l'indicatore digitale, costituito da una cifra da 1 a 8 in prima posizione.
Oltre ai vincoli posti dal tipo di oggetto da identificare, la scelta di usare una codifica piuttosto che un'altra è generalmente dettata dal contesto applicativo.

2.3.1 SGTIN

Il GTIN, come visto, identifica la classe di un prodotto. Qualora si abbia la necessità di identificare univocamente il singolo articolo è necessario usare la codifica *SGTIN* (Serialized GTIN) che si ottiene aggiungendo in coda al codice GTIN un seriale, lungo da 1 a 20 cifre, scelto dall'azienda (solitamente in maniera progressiva).
La codifica SGTIN ricopre un ruolo di rilievo nello standard EPCIS, avendo l'obiettivo di identificare univocamente l'oggetto chiave di un evento EPCIS.

2.4 GDTI

Il codice *GDTI* (Global Document Type Identifier) è la chiave GS1 usata per l'identificazione di documenti fisici, come certificati, fatture, documenti di trasporto, e di documenti elettronici, come immagini digitali o messaggi EDI. È un codice assegnato dall'organizzazione che emette il documento, per facilitarne il riconoscimento, sia per utilizzi interni sia per lo scambio del documento stesso con i partner commerciali [7]. Il codice GDTI è costituito da due parti:

1. una parte fissa, obbligatoria, di 13 cifre che è usata per identificare la tipologia del documento emesso e l'organizzazione emettente; più precisamente, essa è strutturata nel seguente modo:

- un prefisso aziendale GS1, lungo 9 cifre;
- un identificativo del tipo di documento, lungo 3 cifre, assegnato dall'azienda che emette il documento;
- la cifra di controllo;

2. una parte seriale facoltativa costituita da 1 a 17 caratteri alfanumerici, la quale permette di identificare il singolo documento di una certa tipologia.

2.5 SSCC

Il codice GS1 *SSCC* (Serial Shipping Container Code) è la chiave GS1 utilizzata per l'identificazione delle unità logistiche. L'unità logistica è un raggruppamento di unità commerciali confezionate insieme per la gestione del magazzino o per consentire il trasporto delle merci, come per esempio un pallet assemblato da un produttore per evadere un ordine del consumatore [8]. Un codice SSCC ha una lunghezza di 18 cifre e ha la seguente struttura:

- una cifra di estensione che può assumere i valori da 0 a 9;
- il prefisso aziendale GS1, lungo 9 cifre;
- il numero seriale dell'unità logistica, lungo 7 cifre e assegnato sequenzialmente;
- la cifra di controllo.

2.6 GRAI

Il codice GS1 GRAI (Global Returnable Asset Identifier) è la chiave GS1 usata per l'identificazione degli asset riutilizzabili, quindi per identificare pallet, cassette o barili per il trasporto delle merci. Migliora la tracciabilità e i processi di smistamento [9].
Un codice GRAI ha una lunghezza minima di 14 cifre e una lunghezza massima di 30 cifre e ha la seguente struttura:

- una parte fissa obbligatoria lunga 14 cifre che è così costituita:
  - una cifra iniziale che è sempre 0;
  - il prefisso aziendale GS1, lungo 9 cifre;
  - l'identificativo del tipo di asset in un certo prefisso aziendale, lungo 3 cifre;
  - la cifra di controllo.
- una parte opzionale, lunga al più 16 cifre, usata per identificare un singolo asset di una certa tipologia.

2.7 GIAI

Il codice GS1 GIAI (Global Individual Asset Identifier) è la chiave GS1 usata per l'identificazione degli asset individuali di un'azienda [10] quali possono essere, per esempio, un computer, una scrivania, un sensore IoT, un veicolo, eccetera.
Un codice GIAI ha una lunghezza minima di 10 cifre e una lunghezza massima di 39 cifre e ha la seguente struttura:

- il prefisso aziendale GS1, lungo 9 cifre;
- una parte di lunghezza variabile da 1 a 30 cifre, usata per identificare uno specifico asset.

2.8 EPCIS

Lo standard GS1 *EPCIS* (Electronic Product Code Information Services) permette di tracciare lo stato di un prodotto sulla intera filiera produttiva, dal produttore fino a al consumatore finale; consente, inoltre, di condividere tra le varie aziende le informazioni sulla storia del prodotto [11].
EPCIS, in buona sostanza, include entrambe le caratteristiche di tracciabilità e trasparenza tanto importanti all'interno di una supply chain, consentendo di seguire ogni singolo prodotto, a patto che questo sia identificato da un codice SGTIN. Al

momento della stesura di questa tesi di laurea, la versione più recente dello standard è la 2.0 [12].
In figura 2.1, è illustrato il modo in cui EPCIS si inserisce in una tipica infrastruttura IT aziendale.

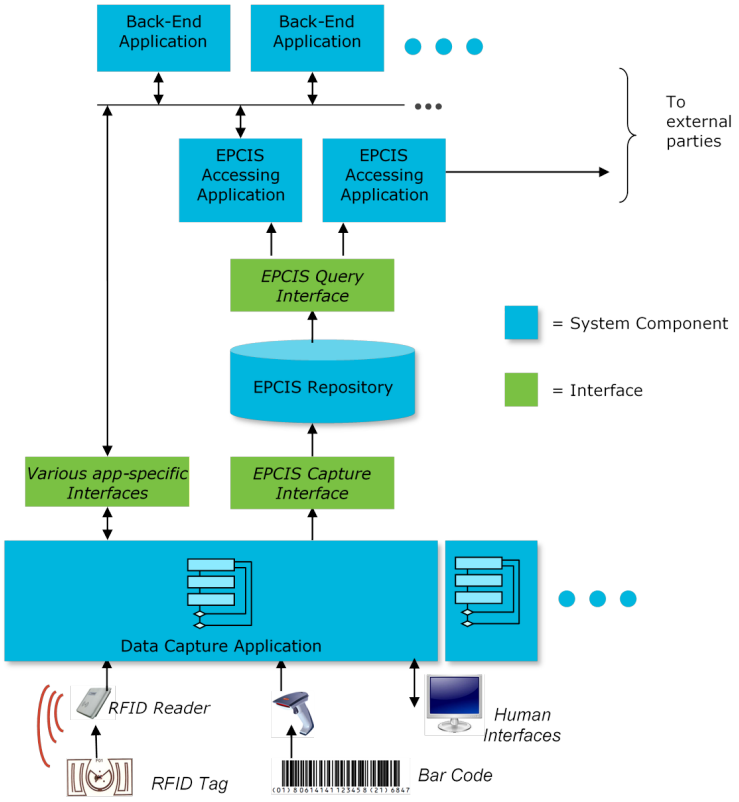

Figura 2.1: tipica collocazione di EPCIS nell'infrastruttura di un'azienda [13].

Lo standard EPCIS definisce:

- l'*Abstract Data Model Layer*, che specifica la struttura (sintattica) di un qualsiasi dato appartenente al dominio EPCIS;

- il *Data Definition Layer*, che specifica quali dati sono scambiabili attraverso EPCIS e cosa ogni singolo dato rappresenta (significato semantico);

- il *Service Layer*, che definisce le due interfaccie EPCIS che una qualsiasi applicazione implementante EPCIS deve esporre ai client affinché questi possano usufruire dello standard. Le due interfacce sono:

1. la *Capture Interface*, attraverso la quale un'applicazione riceve eventi EPCIS che vengono poi in genere memorizzati in un *repository* EPCIS;
2. la *Query Interface*, attraverso cui un'applicazione può inviare un evento EPCIS a un client.

La grande potenza e utilità di EPCIS risiede nella possibilità di aggregare i vari singoli eventi che sono prodotti nel tempo e, attraverso delle analisi, ricavarne informazioni utili secondo paradigmi che possono essere più o meno custom. Alcuni esempi di paradigmi classici sono:

- *tracking*, cioè ricercare l'evento EPCIS più recente per un determinato oggetto, al fine di avere informazioni su dove si trova e sul suo stato;
- *tracing*, cioè ricavare la storia (i.e. la lista di tutti gli eventi) di un oggetto (o di una classe di oggetti) per comprendere, per esempio, il percorso che tale oggetto ha fatto lungo la filiera produttiva;
- *analysis*, cioè analizzare tutti gli eventi che sono avvenuti in un dato luogo o all'interno di un particolare processo aziendale;
- *checking*, cioè confrontare lo stato di un oggetto, in un qualsiasi istante del suo ciclo di produzione, con lo stato che si prevedeva avesse sulla base di una precedente transazione commerciale o di un precedente evento EPCIS che lo riguardava;
- *automation*, cioè il rilevamento in tempo reale di un evento appena generato riguardante una fase di business di interesse.

EPCIS è stato ideato per venire utilizzato congiuntamente con il *CBV* (Core Business Vocabulary), un ulteriore standard GS1 che fornisce un glossario di elementi con i rispettivi valori usabili per popolare le strutture dati e gli eventi definiti da EPCIS [14]. La presenza e l'utilizzo di un dizionario standard e condiviso è un elemento indubbiamente essenziale per poter garantire l'interoperabilità tra i vari sistemi e per uniformare il modo in cui le aziende di tutto il pianeta esprimono determinati elementi o situazioni che riguardano una qualsiasi entità di una supply chain.

2.8.1 Tipologie di eventi

Lo standard EPCIS definisce 5 tipi di evento:

1. l'*ObjectEvent*, che rappresenta ciò che avviene riguardo uno o più oggetti fisici o digitali ed è il tipo di evento più usato, oltre che il più semplice; un suo esempio di utilizzo è la rappresentazione del ricevimento di un prodotto;

2. l'*AggregationEvent*, che rappresenta un evento accaduto a uno o più oggetti che sono o fisicamente aggregati insieme oppure disaggregati l'uno dall'altro; un suo esempio di utilizzo è la rappresentazione di imballi che sono posti su un pallet (aggregazione) o che sono rimossi da quest'ultimo (disagreggazione);

3. il *TransformationEvent*, che rappresenta un evento in cui uno o più oggetti originano un nuovo oggetto. Nel caso di più oggetti, anche per tale tipo di evento essi sono messi insieme come nell'AggregationEvent, con la differenza che il risultato dell'unione non è in questo caso reversibile. Un esempio di utilizzo è la combinazione di farina, mozzarella e pomodoro che danno origine a una pizza;

4. il *TransactionEvent*, che rappresenta un evento in cui uno o più oggetti vengono associati con l'identificativo di una transazione di business. È l'evento meno diffuso, poiché può essere rimpiazzato da un ObjectEvent contenente informazioni sulle transazioni commerciali. Un esempio di utilizzo è l'abbinamento di un prodotto a una fattura commerciale;

5. l'*AssociationEvent*, che è stato introdotto a partire da EPCIS 2.0 e serve per rappresentare quelle situazioni in cui un oggetto è associato (o disassociato) o con un altro oggetto o con un luogo fisico. È simile a un AggregationEvent da cui differisce, però, per due fattori sostanziali:

   (a) l'AggregationEvent rappresenta relazioni oggetto-oggetto, mentre l'AssociationEvent rappresenta anche le relazioni oggetto-luogo;

   (b) l'AssociationEvent è molto adatto a descrivere le relazioni genitore-figlio maggiormente permanenti rispetto alla relazioni descritte dall'AggregationEvent.

EPCIS 2.0 permette di rappresentare gli eventi in formato JSON, mentre nella sua versione antecedente questi potevano essere rappresentati solo in formato XML. La piattaforma iChain ha, tuttavia, fatto uso fin da subito del formato JSON in previsione del cambiamento sopra accennato.
Prima della stesura di questa tesi, la piattaforma iChain era in grado di gestire tutti i tipi di evento a eccezione di AssociationEvent. In questo percorso di tesi si è iniziato a introdurre anche quest'ultima funzionalità.

2.8.2 Le cinque dimensioni

Lo standard EPCIS rappresenta un evento in 5 dimensioni:

1. la dimensione *what*, che rappresenta uno o più oggetti trattati da un evento EPCIS. Esistono due modi differenti per riferirsi agli oggetti:
   (a) *identificazione a livello di istanza*, se l'identificativo associato a un oggetto è un codice GS1 di tipo SGTIN, SSCC o GRAI;
   (b) *identificazione a livello di classe*, se l'identificativo associato a un oggetto è un codice GS1 di tipo GTIN;
2. la dimensione *when*, che contiene una serie di informazioni temporali relative a un evento EPCIS;
3. la dimensione *where*, che contiene informazioni in merito al luogo in cui si è verificato un evento EPCIS. I due principali elementi di questa dimensione sono:
   (a) il punto di lettura (*read point*) che identifica il punto esatto in cui l'evento EPCIS ha avuto luogo ed è spesso il punto in cui il codice a barre di un prodotto è stato letto;
   (b) la posizione di business (*business location*), che indica il luogo in cui si troverà l'oggetto (o gli oggetti) coinvolto nell'evento EPCIS finché non ci sarà un altro evento che lo riguarda;
4. la dimensione *why*, che contiene una serie di informazioni sul contesto di business (*business context*) che caratterizza l'evento EPCIS ed è quindi cruciale

per dare un senso ai dati EPCIS. I principali elementi di questa dimensione sono:

   (a) il *business step*, che denota una specifica attività all'interno del processo aziendale che ha scatenato la generazione (e l'acquisizione) dell'evento EPCIS;

   (b) la *disposition*, che specifica lo stato degli oggetti, nella dimensione what, dell'evento EPCIS dopo che questo si è verificato;

   (c) riferimenti alle transazioni commerciali (*business transaction references*);

5. la dimensione *how*, definita a partire dallo standard 2.0 di EPCIS, fornisce informazioni su come l'evento EPCIS è stato acquisito e perciò, nello specifico, contiene informazioni sui *sensori* usati per catturare l'evento.

La piattaforma iChain, al momento della stesura di questa tesi, gestisce tutte le dimensioni tranne la dimensione *how*.

2.8.3 Master data

EPCIS, come visto, fornisce una serie di identificativi il più possibile generici, definiti dal CBV, per rappresentare le varie informazioni (prodotti, luoghi, documenti, eccetera) che possono riguardare un evento; inoltre, i vari identificativi sono una serie di codici GS1. Di conseguenza, i dati forniti da EPCIS, detti *visibility data*, non sempre sono sufficienti a identificare uno specifico contesto operativo. Per tale ragione, lo standard prevede la possibilità di inserire all'interno dell'evento EPCIS i cosiddetti *master data*, che altro non sono che una lista di coppie chiave-valore personalizzabili inseribil in un campo dell'evento chiamato *ilmd* (Instance/Lot master data). Si consideri per esempio un evento riguardante l'acquisto di un cavallo. Tale cavallo sarà rappresentato, nella dimensione what, da un codice SGLN che non è però in grado di fornire informazioni riguardo, per esempio, il colore o la razza del cavallo e lo standard EPCIS/CBV non fornisce alcun vocabolo atto a rappresentare queste informazioni in quanto, appunto, non sono per nulla generiche. Ecco che tali informazioni potrebbero essere rappresentate nei master data.

3

Estensione di alcune funzionalità di iChain

Uno degli obiettivi di questa tesi è stato quello di estendere alcune delle funzionalità già disponibili in iChain. Lo scopo del presente capitolo è di illustrare come operavano alcune funzionalità di iChain prima di venire estese e come queste, poi, sono state migliorate e perché.

3.1
Le regole di inoltro

Per poter descrivere il primo lavoro di cui ci si è occupati con la presente tesi, è necessario introdurre brevemente cosa sono le *regole di inoltro*.
Come si è detto nel capitolo introduttivo, la *trasparenza* è molto importante all'interno di una *supply chain*. Tuttavia, attualmente le aziende sono particolarmente interessate a mantenere la riservatezza dei loro dati, i cosiddetti "segreti industriali". Le aziende sono consapevoli dei benefici che si hanno da una collaborazione con le altre aziende ma, ovviamente, non sono affatto disposte a rinunciare né alla possibilità di decidere a chi dare una visione sui propri dati/eventi (quindi con chi essere trasparenti) né alla possibilità di condividere solo la porzione dei propri dati che ritengono conveniente e non pericolosa per preservare i propri segreti industriali. La piattaforma iChain, certamente consapevole di questo orientamento del mondo aziendale, viene totalmente incontro a queste esigenze dettate dal mondo degli affari proprio grazie alle *regole di inoltro*, che costituiscono uno degli aspetti chiavi della piattaforma sin dalle sue origini. Il tool per la creazione di regole di inoltro di iChain consente agli utenti registrati sulla piattaforma di scegliere liberamente e senza limiti quali dati condividere con i propri partner e quali mantenere privati, ottenendo così la *granularità* di trasparenza desiderata. Per ogni partner si può ovviamente specificare una (o più) particolare regola di inoltro, grazie all'elevato livello di personalizzazione che la piattaforma fornisce.

Dunque, una volta che un'azienda A definisce una regola di inoltro dei propri eventi EPCIS verso un'azienda B, l'azienda B avrà accesso alle informazioni fornite da A. Ciò si realizza duplicando i dati degli eventi EPCIS che si trovano nel database dell'azienda A sul database degli eventi EPCIS dell'azienda B, con la specificazione che quegli eventi sono stati generati da A e non da B; ciò si ottiene indicando semplicemente l'azienda A nel campo *data_ origin* (sorgente dei dati) del data model che rappresenta l'evento EPCIS copiato nel database di B. Si specifica che tale campo non era originariamente nella piattaforma presente ed è stato appositamente inserito nel corso della presente tesi (come si vedrà nel paragrafo 4.5).
Questo meccanismo, apparentemente perfetto, presentava tuttavia un punto debole del quale si tratterà compiutamente nel prossimo paragrafo.

3.2
Il super admin e l'accesso ai domini

In questa tesi, è stato aggiunto alla piattaforma iChain un utente speciale, avente il ruolo di *super-amministratore*. Il super-amministratore è utile per la risoluzione di seri problemi, sia del sistema sia dei vari utenti, o per effettuare manutenzione di sistema. Inoltre, usufruendo di privilegi elevati, i più alti all'interno della piattaforma, il super-amministratore ha sempre un accesso, senza la minima restrizione (fatta eccezione per le password, in quanto memorizzate sotto forma di hash), a qualunque dato presente sulla piattaforma stessa, indipendentemente da chi sia il vero proprietario di tale dato (ovviamente l'accesso a tali dati è possibile solo su richiesta del proprietario dei dati, che per esempio potrebbe incaricare iChain per effettuare delle analisi dei propri dati, o su richiesta degli organi giudiziari). Indubbiamente il super-amministratore non ha mai avuto problemi nell'accedere al database della piattaforma e ai dati in essa contenuti. Tuttavia, qualora il super-amministratore avesse voluto accedere ai dati di qualche azienda tramite l'interfaccia web di iChain, per poter così usufruire di alcuni dei tool grafici che essa offre e la cui visualizzazione grafica è estremamente importante per effettuare alcune analisi dei dati stessi, si presentava un problema: i vari endpoints della piattaforma, restituiscono i dati (template di eventi, eventi EPCIS, eccetera) selezionando dal database quelli che,

nel rispettivo data model, hanno come valore nel campo *object_ owner* (proprietario dei dati) il dominio dell'utente che ha fatto la richiesta al server; di conseguenza il super-amministratore non avrebbe mai potuto ricevere alcun dato in quanto il dominio con cui è registrato sulla piattaforma non ne contiene alcuno.
Una possibile soluzione a questo problema avrebbe potuto essere quella di copiare tutti i dati di tutte le compagnie registrate su iChain e di cambiare in ciascuno il campo *object_ owner*, assegnandogli il dominio del super-amministratore. Per una piattaforma che non vuole essere soltanto un progetto accademico, bensì un'applicazione di grande rilevanza quale ambisce a diventare iChain, tale soluzione non potrebbe essere applicata poiché la mole di dati da copiare sarebbe potenzialmente enorme e ciò richiederebbe, di fatto, una grossa quantità di byte per la copia dei dati dai domini dei vari utenti registrati alla piattaforma al dominio dell'utente con i privilegi di super-amministratore. Il problema principale sarebbe stato soprattutto la copia dei dati degli eventi EPCIS, in quanto una sola azienda può potenzialmente produrre milioni di eventi al giorno; è proprio questo il difetto delle regole di inoltro: un'azienda B è interessata a ricevere i dati di un'azienda A per potervi accedere e analizzarli, ma nel caso di iChain tale copia sarebbe servita solo per poter visualizzare i dati tramite l'interfaccia grafica, in quanto iChain di fatto possiede già gli eventi delle varie aziende nel proprio database. Più volte in questa tesi si è parlato di come le varie aziende facciano le loro scelte anche (o soprattutto) in considerazione dei costi che tali scelte richiedono e questo sarebbe stato un costo importante per iChain stessa.
Si è quindi operata una seconda scelta, più efficace della precedente (anche in termini economici), che ha sì richiesto un piccolo costo in termini di tempo per modificare parte del codice già esistente, ma consente all'utente con i privilegi di super-amministratore di accedere ai dati di qualunque azienda registrata su iChain senza dover effettuare la copia di alcun dato. Tale scelta è stata quella di modificare la maggior parte degli endpoints esposti da iChain affinché essi ricevano anche il dominio di cui si vogliono ottenere i dati: in questo modo ogni endpoint non utilizza più il dominio dell'utente connesso come valore da assegnare al campo *object_ owner* per filtrare i dati da restituire, ma utilizza il dominio che gli viene passato. L'utente super-amministratore può, così, accedere ai dati di qualunque azienda semplicemen-

te passando a un endpoint il dominio dell'azienda stessa.

Se ci si fosse limitati a risolvere il problema in questo modo, senza aggiungere altro, iChain avrebbe da un lato risparmiato risorse, in quanto non avrebbe dovuto allestire dei data-store per contenere semplici copie necessarie solo per permettere all'utente con i privilegi di super-amministratore di accedere ai dati tramite l'interfaccia grafica, ma dall'altro ne avrebbe dovuti spendere molti di più per rimborsare i propri clienti che si sarebbero visti privare della segretezza dei propri dati: chiunque, e non solo il super-amministratore, avrebbe potuto accedere infatti ai dati di un'azienda semplicemente invocando un endpoint e passandogli il dominio di quell'azienda. Quindi, oltre a modificare gli endpoints affinché essi possano ricevere il dominio di cui si vogliono ottenere i dati, gli endpoints verificano anche che l'utente che sta usando uno specifico endpoint fornito da iChain sia autorizzato ad accedere ai dati del dominio richiesto, non restituendo nulla qualora non si fosse autorizzati. Il meccanismo di autorizzazione adottato è semplice ma efficace: l'utente che invoca l'endpoint, che è noto in quanto può dimostrare la propria identità mandando al server un token JWT fornitogli dal server stesso durante la fase di login, può accedere a un dominio solo se esso appartiene alla lista dei domini a cui ha accesso quell'utente (il data model di ogni utente contiene una lista di domini a cui l'utente ha accesso), a eccezione ovviamente del solo utente con i privilegi di super-amministratore che può accedere a qualunque dominio pure se questo non è presente nella sua lista di domini.

Di seguito, si riporta la funzione implementata in questa sede per la verifica dei domini, che come si può vedere è estremamente stringata e per nulla complessa ma è di fondamentale importanza:

```
1 async def can_user_access_to_domain(domain: str, current_user: User
      = Depends(get_current_active_user)):
2     if Permissions.super_admin not in current_user.rights:
3         if domain not in current_user.active_domains:
4             raise HTTPException(status_code=status.
                  HTTP_403_FORBIDDEN, detail="Not authorized")
```

3.3

Configuratore del flusso produttivo

L'insieme dei vari flussi produttivi di un'azienda costituisce, di fatto, la rappresentazione delle varie attività che l'azienda svolge. Il *flusso produttivo*, infatti, rappresenta tutte quelle sequenze di operazioni e di fasi che permettono di realizzare un determinato prodotto, indicando tutta una serie di eventi, ognuno dei quali identifica un particolare stato di quel prodotto fino alla sua forma definitiva. A volte un flusso produttivo può essere molto complesso e ramificato, ma è comunque importante che esso sia ben definito per consentire a un'azienda di minimizzare, a parità di quantità e qualità del prodotto ottenuto, i costi di produzione oppure, alternativamente, di massimizzare la produzione a parità di costi. Per tale ragione, uno dei tool di iChain è il *production flow configurator* (configuratore del flusso produttivo), un editor grafico che consente di creare una serie di progetti in cui i vari flussi produttivi vengono definiti graficamente. Tale tool, per ogni progetto creato permette di:

- inserire dei nodi rappresentanti un evento EPCIS o, per meglio dire, un *template di evento EPCIS*[1];
- collegare questi nodi fra loro tramite degli archi entranti/uscenti;
- creare e inserire dei nodi di bozze di template di eventi;
- completare i nodi di bozze di template di eventi rendendoli veri e propri eventi EPCIS;
- cancellare i nodi di bozze di template di eventi;

In figura 3.1, è mostrato un esempio di flusso produttivo realizzato con l'ausilio del *production flow configurator di iChain.*

[1]Per ulteriori informazioni sui template di eventi, consultare l'appendice A.

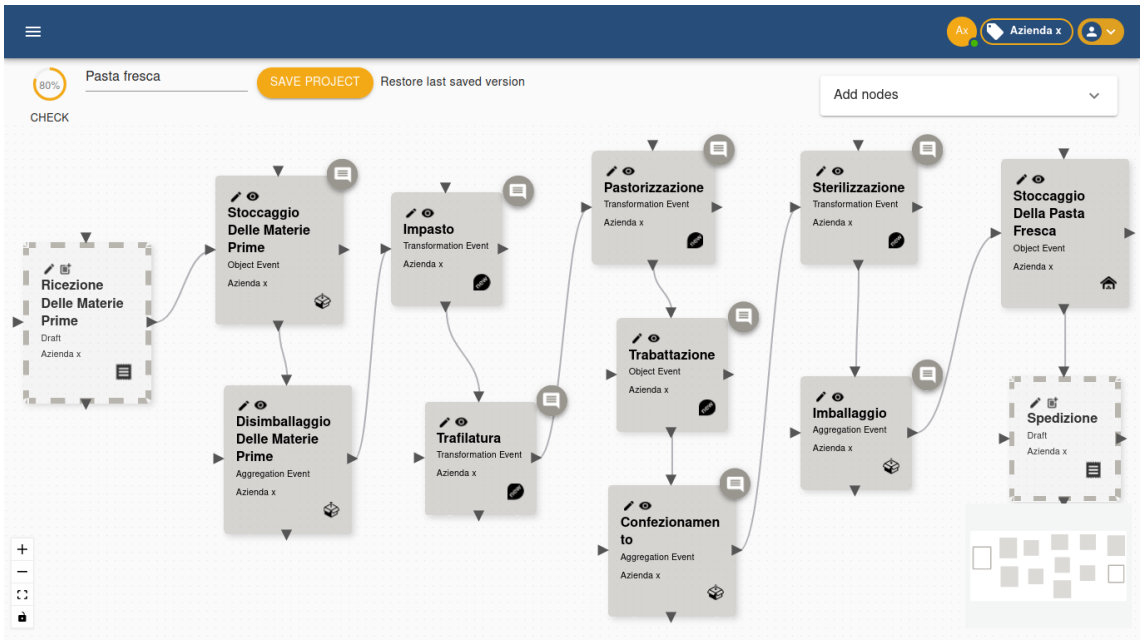

Figura 3.1: esempio di descrizione del processo produttivo della pasta fresca realizzato con il tool *production flow configurator* su iChain (i riquadri tratteggiati simboleggiano nodi di template bozza di eventi.

Siccome un flusso produttivo è spesso ideato sulla base di specifiche condizioni di ambiente (e.g. capitale, concorrenza, brevetti, domanda) e delle informazioni effettivamente a disposizione, non sempre chi è incaricato di progettare e disegnare il processo produttivo può avere in mente fin da subito tutti i dettagli degli eventi EPCIS che lo coinvolgono: si è quindi deciso di estendere l'attuale tool *production flow configurator* di iChain aggiungendo la possibilità di inserire dei *nodes of draft event template* (nodi di bozze di template di eventi), cioè nodi di cui è sufficiente riportare, in fase di creazione degli stessi, solo il nome del template dell'evento e un'eventuale descrizione. Ogni bozza di un template di evento potrà poi essere completata (diventando, così, un template definitivo di evento EPCIS) in un secondo momento, quando saranno meglio definiti i dettagli dell'evento EPCIS che esso intende rappresentare. Inoltre, è fornita la possibilità di eliminare una bozza di un template di evento.

3.3.1 Progettazione dei nodi di un template bozza di evento

Come primo passo per l'introduzione dei nodi bozza e dei template bozza, è stato necessario migliorare il data model rappresentante un template di evento e il data

model rappresentante un progetto creato con il *production flow configurator*. Tali modelli sono rispettivamente chiamati *capture event template* e *graphic configurator*. Alla struttura originaria del data model *capture event template*, che può essere consultata per intero nell'appendice A, è stato aggiunto il campo *is_ draft* che altro non è che un semplice flag booleano: se è true, indica che il template di evento è ancora in uno stato di bozza, mentre se è false indica che il template di evento è stato completato. Tale flag ha lo scopo di consentire, all'interno della piattaforma iChain, la distinzione tra un template di evento completo e uno che ancora non lo è, consentendo dunque di identificare le bozze e impedire che queste possano essere usate in alcune operazioni svolte dalla piattaforma (come ad esempio la generazione di un evento EPCIS a partire da un template di evento) che richiedono, invece, che il template di evento EPCIS sia completo. Inoltre, il data model *capture event template* ha originariament il campo *version*, il quale contiene la versione del template; nel caso di template completi, tale campo parte da 1 quando è creato ed è incrementato di 1 ogni volta che subisce delle modifiche. Quando vengono creati template bozza, invece, si è scelto di inizializzare tale campo a 0 e ogni modifica al nome o alla descrizione del template bozza non incrementeranno la versione; in seguito al completamente del template bozza, infine, il campo *version* diventerà uguale a 1 e da questo momento assumerà gli incrementi unitari indicati per il campo *version* dei template completi.
Il data model *graphic configurator*, invece, contiene i seguenti campi:

- *project_ name*, che è il nome di un progetto creato con il *production flow configurator*;

- *creator*, che contiene lo username dell'utente che ha creato il progetto;

- *last_ editor*, che contiene lo username dell'utente che ha effettuato l'ultima modifica al progetto (si fa presente, infatti, che ogni azienda registrata su iChain può avere naturalmente più di un utente);

- *last_ update _ time*, che altro non è se non la data in cui è stata fatta (e ovviamente salvata) l'ultima modifica al progetto;

- *counter _draft _nodes*, che indica il numero di nodi all'interno di un progetto che rappresentano una bozza di template di evento;

- *counter _not _draft _nodes*, che indica il numero di nodi all'interno di un progetto che rappresentano un template di evento completo;

- *nodes*, che è una lista contenente tutti i nodi all'interno del progetto. Il data model associato a ogni nodo presenta, a sua volta, i seguenti attributi:

  - *type*, che rappresenta il tipo di nodo. Al momento l'unico valore possibile per tale parametro è *custom node*, l'unica tipologia di nodo che la piattaforma gestisce attualmente;

  - *position*, che contiene le coordinate x e y in cui si trova il nodo sullo schermo;

  - *node _data*, che contiene informazioni sull'azienda proprietaria del nodo e sul template che esso rappresenta. Al momento, l'unica informazione sull'azienda è il dominio principale con cui essa è registrata su iChain, mentre per quanto riguarda le informazioni sul template di evento attualmente sono limitate alle due seguenti:

    * *template _id*, che è contiene l'id del modello che il nodo rappresenta;
    * *is_ draft*, che indica se il template di evento rappresentato dal nodo è una bozza o meno;

- *edges*, che è una lista contenente tutti gli archi all'interno del progetto, dove per ciascun arco si indicano il nodo uscente e il nodo entrante.

Tra i campi suddetti, quelli aggiunti al data model *graphic configurator* in questo percorso di tesi sono stati: *counter _draft _nodes*, *counter _not _draft _nodes*, *is_ draft*. Tali campi sono stati inseriti per facilitare l'implementazione lato front-end della piattaforma. Infatti, tramite il campo *is_ draft* presente nel data model di ogni nodo nel progetto il front-end è in grado di sapere istantaneamente se un nodo rappresenta un template bozza di evento o un template completo di evento, senza dover per ciascun nodo richiedere al server ulteriori informazioni sul template

di evento a partire dai loro id; in tal modo è così possibile offrire una rappresentazione grafica opportuna. In figura 3.1 vi sono due nodi di template bozza, il primo e l'ultimo (ovviamente qualsiasi nodo all'interno del flusso produttivo può contenere un template bozza). Invece, tramite i campi *counter _draft _nodes*, *counter _not _draft _nodes* il front-end può agevolmente calcolare la percentuale di completamento del progetto, cioè la percentuale che indica quanti sono i nodi di template completi di eventi rispetto al totale, senza dover effettuare il conteggio per conto proprio. Un esempio di percentuale di completamente del progetto è visibile in alto a sinistra nella figura 3.1.
Ogni qual volta un template bozza di evento viene completato, però, tali semplificazioni lato front-end hanno introdotto, lato back-end, la necessità di dover operare in modo transazionale sulle due collezioni del database di iChain contenenti, una, i vari template di eventi e, l'altra, i vari progetti del *production flow configurator*, in quanto a seguito del completamento di un template è necessario anche aggiornare in ogni progetto i due contatori e il campo *is_ draft* presente nel data model di ogni nodo rappresentante il template ultimato (questo perché, ovviamente, anche se un template di evento viene completato agendo da uno specifico progetto, tal completamento deve risultare visibile a ogni progetto contenente almeno un nodo per quel template di evento).
Come sopra evidenziato, inoltre, i nodi contenenti template bozza possono essere rimossi da un progetto, fintantoché il template bozza non viene completato. L'eliminazione da tutti i progetti di tutti i nodi contenenti uno specifico template bozza comporta l'eliminazione definitiva di quest'ultimo.
Per tale parte di lavoro è stato anche necessario implementare i seguenti endpoints REST:

- l'endpoint POST /api/capture/event_templates/drafts/by_domain/, che consente di creare un template bozza di evento per un dato domino aziendale. L'endpoint richiede in input:

    1. nome che dovrà avere quel template bozza. Tale campo è obbligatorio e deve essere univoco all'interno del dominio aziendale (cioè un dominio

aziendale non può avere due template con lo stesso nome, indipendentemente dal fatto che siano delle bozze o meno);

2. una descrizione opzionale del template bozza.

Se è stato possibile creare il template bozza, l'endpoint restituirà al client il suo id;

- l'endpoint PUT /api/capture/event_templates/drafts/_id, che consente di aggiornare il nome o la descrizione di un template bozza a partire dal suo id. Oltre a dover verificare che esista veramente un template avente quell'id e che tale template sia una bozza, è anche necessario che il client che invoca tale endpoint abbia i permessi per modificare il template bozza, ovvero che sia il proprietario di tale template. Anche in questo caso, il nuovo nome fornito non deve già esistere all'interno del domino aziendale;

- l'endpoint /api/capture/event_templates/drafts/finalize/, che permette di completare un template bozza ricevendo in input l'id del template e i vari campi del template stesso, aggiornando di conseguenza i vari progetti aventi nodi rappresentanti tale template. Tale endpoint, oltre a effettuare tutti i controlli del precedente endpoint, deve anche verificare la validità di tutti i campi del template forniti in input.

È stato poi necessario apportare modifiche ai seguenti endpoints REST già esistenti:

- l'endpoint POST /api/capture/epcis_events/simplified/, che consente di creare un evento EPCIS a partire dai dati forniti da un template, è stato modificato per rifiutare quei template che sono ancora bozze;

- gli endpoints /api/capture/event_templates/ e /api/capture/event_templates/_by_domain/, che restituiscono rispettivamente tutti i template di eventi della piattaforma o tutti i template di eventi di un dato dominio, sono stati modificati per restituire anche il campo *is_draft*;

- l'endpoint /api/graphic_conf/project/by_name/event_graph/plot/, che restituisce un flusso di progetto dato il suo nome, è stato modificato per restituire anche i nuovi campi aggiunti al data model *graphic configurator*.

3.4
Supporto all'Association Event

Come si è visto nel capitolo 2, a partire dallo standard EPCIS 2.0 è stata introdotto una quinta tipologia di evento EPCIS chiamata *Association Event*.
Essendo la piattaforma iChain fortemente incentrata sullo standard EPCIS, essa non poteva in alcun modo non fornire a sua volta supporto per questa nuova tipologia di evento.

3.4.1 AssociationEvent: cos'è e perché è necessario

Si è precedentemente detto che un *AssociationEvent* descrive l'associazione o la dissociazione tra uno o più oggetti fisici con un altro oggetto fisico o con una specifica località fisica. Si è anche evidenziato che tale evento è simile all'*AggregationEvent*, da cui comunque differisce per i seguenti aspetti:

1. con l'*AssociationEvent*, gli oggetti possono essere associati anche con luoghi fisici;

2. l'*AssociationEvent* rappresenta associazioni maggiormente permanenti.

Si cercherà ora di far capire con un esempio perché, al di fuori delle relazioni oggetto-luogo, l'introduzione dell'*AssociationEvent* è importante anche per le relazioni oggetto-oggetto. Prima, però, si indicheranno tre fra i campi contenuti in un evento *AggregationEvent*, utili per meglio comprendere l'esempio:

1. il campo *parentID*, che contiene l'oggetto genitore della relazione identificato a livello di istanza;

2. il campo *childEPCs*, che contiene la lista degli oggetti figli della relazione, ciascuno dei quali è identificato a livello di istanza;

3. il campo *action*, che può assumere i valori *ADD* o *DELETE* se sta rispettivamente avvenendo una aggregazione o una disaggregazione tra l'oggetto contenuto nel campo *parentID* e gli oggetti contenuti nel campo *childEPCs*. Tale campo può assumere anche il valore *OBSERVE* se si sta solo osservando

il contenuto dell'aggregazione (quest'ultimo caso non è interessante per l'esempio che si farà). Siccome prima che possa avvenire una disaggregazione deve ovviamente esserci una aggregazione, se *action* = *ADD* (aggregazione) il campo *childEPCs* non può essere una lista vuota; se *action* = *DELETE* (disaggregazione), invece, il campo *childEPCs* può anche essere una lista vuota. In tal caso, tutti gli oggetti precedentemente aggregati saranno disaggregati, altrimenti, nel caso in cui la lista non sia vuota, solo gli oggetti in essa specificati saranno disaggregati dal genitore della relazione.

Detto ciò, è ora possibile procedere con l'esempio.
Si immagini, innanzitutto, di avere un camion, all'interno del quale viene in seguito installato un termostato che dovrà persistere nel camion per molto tempo. Questa prima situazione può essere rappresentata con un evento EPCIS di tipo *AggregationEvent*: siccome sta avvenendo una aggregazione il campo *parentID* farà riferimento al camion, la lista *childEPCs* conterrà il termostato e il campo *action* conterrà il valore *ADD*. In seguito, all'interno del camion vengono inserite anche 1000 casse di frutta. Anche questa seconda situazione può essere rappresentata con evento *AggregationEvent*; anche stavolta sta avvenendo un'aggregazione, dunque il campo *parentID* farà riferimento al camion, la lista *childEPCs* conterrà le 1000 casse di frutta e il campo *action* conterrà il valore *ADD*. Più avanti ancora nel tempo, magari dopo altri vari eventi EPCIS non di interesse per il il nostro esempio, giunge il momento di rimuovere le 1000 casse di frutta dal camion. Quest'ultimo caso è rappresentabile sempre con un evento *AggregationEvent*; stavolta però sta avvenendo una disaggregazione, quindi nel *parentID* si farà riferimento al camion, il campo *action* conterrà il valore *DELETE* e poi si vorrebbe auspicabilmente che il campo *childEPCs* contenesse una lista vuota per indicare di rimuovere tutte le 1000 casse di frutta. Tuttavia, ciò non è fattibile in quanto, se il campo *childEPCs* contenesse una lista vuota, oltre alle 1000 casse di frutta anche il termostato sarebbe disaggregato dal camion. Bisognerebbe quindi inserire nella lista *childEPCs* gli identificativi di tutte le 1000 casse di frutta, causando così un aumento della lunghezza dell'evento EPCIS che comporterebbe un maggiore spazio richiesto per memorizzarlo in un database e una minore leggibilità (in questo secondo caso, infatti, un qualsiasi

utente dovrebbe leggere tutti gli ID per capire che l'intenzione dell'evento è quella di rappresentare la rimozione di tutte le 1000 casse di frutta dal camion).
Ecco, quindi, che l'utilizzo di *AssociationEvent* torna in questo caso utile. Se nel primo evento dell'esempio, ossia quando il termostato è installato nel camion, si usasse un *AssociationEvent* al posto di un *AggregationEvent* il problema sarebbe risolto e sarebbe così possibile avere una lista vuota nel campo *childEPCs* nell'ultimo evento dell'esempio: in questo modo, infatti, la disaggregazione dal camion riguarderebbe solo le 1000 casse di frutta ma non il termostato.
Pertanto, quando l'oggetto genitore di una relazione potrebbe essere soggetto a successive relazioni più temporanee, è conveniente usare l'*AssociationEvent* per le relazioni di più lunga durata e l'*AggregationEvent* per quelle temporanee.

3.4.2 Aggiunta dell'AssociationEvent su iChain

Seppure, come si è visto, l'*AssociationEvent* è utilizzabile in situazioni diverse da un *AggregationEvent*, i campi specificati dallo standard EPCIS, contenuti nelle due tipologie di eventi, sono gli stessi (e.g. anche l'*AssociationEvent* ha i campi *parentID*, *childEPCs*, *action*) e hanno quasi completamente la stessa semantica. L'unica differenza semantica è rappresentata dal campo *parentID* che, nel caso di un *AssociatonEvent*, come si è detto può contenere, oltre a un *objectID* (identificativo di un oggetto a livello di istanza) anche una *locationID* (identificativo di un luogo, cioè un GLN o un SGLN). Non è quindi stato necessario modificare sostanzialmente il data model originale di evento EPCIS, che continua a contenere i campi che aveva prima, se non per il fatto che all'insieme dei valori, noti a priori, ammissibili per il campo *type* del data model di un evento EPCIS, è stato aggiunto anche il valore *AssociationEvent*. Si mostra di seguito il breve frammento di codice contenente l'insieme dei valori possibili per il campo *type*.

```
1 class EPCISEventTypeEnum(str, Enum):
2     EMPTY_EVENT = ""
3     OBJECT_EVENT = "ObjectEvent"
4     AGGREGATION_EVENT = "AggregationEvent"
5     TRANSACTION_EVENT = "TransactionEvent"
6     ASSOCIATION_EVENT = "AssociationEvent"
```

È possibile notare che questa enum contiene anche il valore *EMPTY_EVENT*. Tale valore non è ovviamente presente nello standard EPCIS, in quanto un evento EPCIS deve ineludibilmente avere un tipo associato che lo rappresenti. All'interno della piattaforma iChain, tale valore è stato introdotto unicamente come valore da assegnare al tipo di evento di un template bozza di evento EPCIS.

Per supportare la tipologia di evento *AssociationEvent* all'interno di iChain è stato, però, necessario modificare il meccanismo di validazione del campo *parentID*, affinché tale campo sia sì in grado di contenere anche una *locationID* ma se e solo se il tipo di evento è appunto un *AssociationEvent*.

Come primo passo necessario per introdurre su iChain il tipo di evento *AssociationEvent*, è stato aggiunto al data model di un evento EPCIS una funzione etichettata con un *decorator* (che semanticamente è equivalente a un'annotazione di Java) di tipo *validator* per il campo *parentID* del data model (iChain fa e faceva ampio uso della libreria *Pydantic*[15], cioè una libreria python che fornisce meccanismi per la costruzione dei modelli di dati, la loro configurazione e validazione). Di seguito si riporta il frammento di codice (in una versione leggermente semplificata):

```
1  @validator('parentID')
2  def parent_id_validity(cls, event_parent_id, values, **kwargs):
3      e_type = values['type']
4      if event_parent_id is not None:
5          if e_type == EPCISEventTypeEnum.OBJECT_EVENT or e_type ==
               EPCISEventTypeEnum.TRANSFORMATION_EVENT:
6              raise ValueError(f"{e_type}(s) don't have a parentID
                   field")
7          if e_type == EPCISEventTypeEnum.TRANSACTION_EVENT or e_type
               == EPCISEventTypeEnum.AGGREGATION_EVENT:
8              parent_id_validator(event_parent_id, False)
9          else:
10             parent_id_validator(event_parent_id, True)
11     else:
12         if e_type == EPCISEventTypeEnum.ASSOCIATION_EVENT or e_type
               == EPCISEventTypeEnum.AGGREGATION_EVENT:
13             raise ValueError(f"{e_type}(s) must have a parentID
                   field")
14     return event_parent_id
```

Tale frammento di codice svolge due compiti:

1. verifica la presenza del campo *parentID* all'interno dell'evento EPCIS: tale campo deve essere obbligatoriamente presente per gli eventi di tipo *AssociationEvent*, *AggregationEvent* e *TransactionEvent* mentre non deve essere presente negli eventi di tipo *ObjectEvent* e *TransformationEvent*;

2. per gli eventi di tipo *AggregationEvent* e *TransactionEvent*, verifica che il campo *parentID* contenga un *objectID* valido, mentre per gli eventi di tipo *AssociationEvent* verifica che il campo *parentID* contenga un *objectID* valido o una *locationID* valida. Ciò è fatto invocando la funzione *parent_id_validator*, passandole come primo parametro il *parentID* da validare e, come secondo parametro, il valore booleano rispettivamente *False* o *True*.

Di seguito si riporta la funzione *parent_id_validator*:

```
1  def parent_id_validator(parent_id: str, verify_location_id: bool):
2      result: bool = True
3      if parent_id.startswith("urn:gs1:gln:") and verify_location_id:
4          check = check_gln(parent_id.split(":")[-1])
5          if not check:
6              raise ValueError('Invalid GLN format')
7      elif parent_id.startswith("urn:epc:id:sgln:") and
           verify_location_id:
8          check = check_sgln(parent_id)
9          if not check:
10             raise ValueError('Invalid SGLN format')
11     elif parent_id.startswith("http://"):
12         internal_type = parent_id.split("/")[4]
13         if verify_location_id and internal_type != "obj" and
               internal_type != "loc":
14             raise ValueError("Invalid format")
15         elif not verify_location_id and internal_type != "obj":
16             raise ValueError("Invalid format")
17     else:
18         result = gs1_item_validator(parent_id)
19     return result
```

Il secondo argomento (parametro) della funzione, il *verify _location_id*, che è di tipo booleano, serve a indicare se è necessario verificare anche che il *parentID* possa eventualmente essere una *locationID* oltre che un *objectID*.
Il secondo passo necessario per introdurre su iChain il tipo di evento *AssociationEvent* è modificare i due algoritmi più importanti programmati su iChain, ovvero:

- il *history diagram algoritm*, che ha il compito di calcolare tutte le strutture dati necessarie per visualizzare graficamente la storia di un dato prodotto a livello di filiera;

- il *marble diagram algoritm*, che ha il compito di calcolare le strutture dati necessarie a rappresentare graficamente il flusso produttivo globale (ovvero riguardante qualsiasi prodotto), a livello di filiera, in funzione di tutti gli eventi EPCIS che vengono generati nel tempo, tenendo naturalmente conto delle *regole di inoltro* definite.

Questi due algoritmi devono essere in grado di produrre le opportune strutture dati necessarie per visualizzare i rispettivi diagrammi anche per gli eventi di tipo *AssociationEvent* che hanno come *parentID* una *locationID* (nel caso di un *objectID* non c'è alcun problema al momento noto).

3.5
Richieste di partnership

Come si è visto, iChain fornisce le *regole di inoltro* affinché ogni azienda possa definire cosa condividere con terzi dei propri dati. Nel paragrafo 3.1 si è visto come la condivisione di eventi EPCIS da un'azienda A a un'azienda B si traduce nel copiare nel database dell'azienda B gli eventi EPCIS che l'azienda A ha condiviso con essa. Analogamente un'azienda A può condividere le informazioni relative ai propri prodotti o ai propri luoghi a un'azienda B, che si traduce sempre in una copia di tali dati da un database a un altro. Nella figura 3.2 si mostra un esempio del tool grafico che iChain mette a disposizione per creare delle *regole di inoltro*.

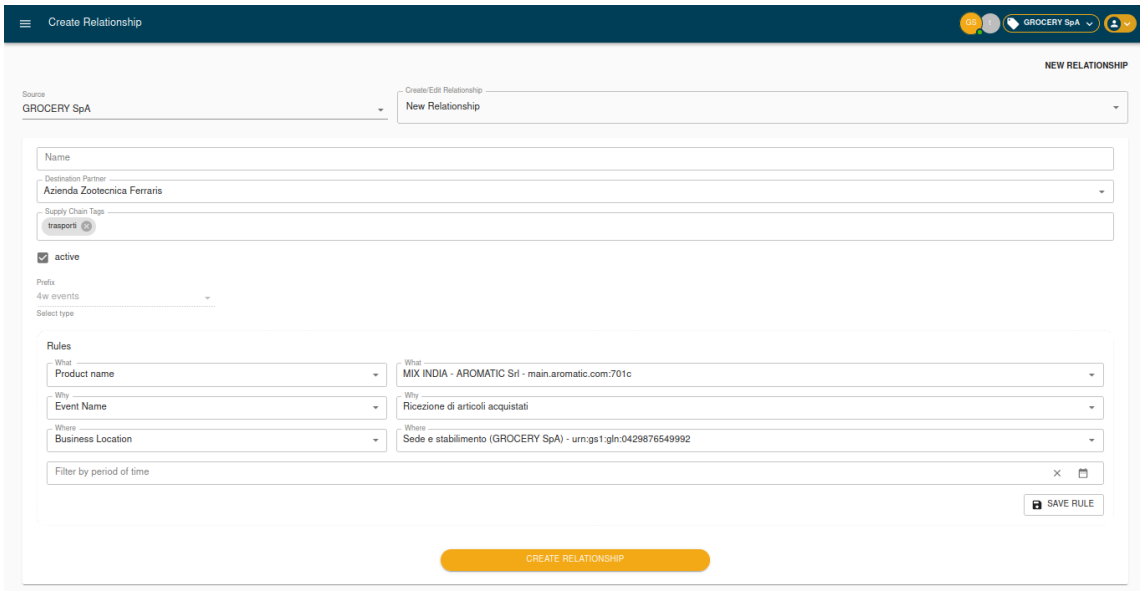

Figura 3.2: esempio di creazione di una *regola di inoltro* riguardante sia un prodotto, sia un evento, sia un luogo, personalizzabile attraverso il relativo tool grafico di iChain.

Antecedentemente all'elaborazione della presente tesi, qualsiasi azienda iscritta su iChain poteva definire una *regola di inoltro* nei confronti di una qualsiasi altra azienda a sua volta registrata sulla piattaforma senza tenere conto del concetto di *relazione/rapporto commerciale*; eppure, nel mondo degli affari, qualsiasi forma di collaborazione tra due aziende è in genere preceduta da una serie di colloqui tra le parti coinvolte, con l'obiettivo di definire delle condizioni e degli obblighi che soddisfino entrambe le parti interessate a seguito dei quali ne scaturisce, successivamente, una concreta collaborazione tra le due aziende.

In considerazione di quanto detto, si è deciso di introdurre su iChain il concetto di *partnership request* (i.e. *richiesta di collaborazione* o *richiesta di partnership*). Un'azienda A potrà ora inviare a un'azienda B una richiesta di collaborazione e solo in seguito a una risposta affermativa da parte dell'azienda B vi sarà una vera e propria collaborazione tra le due aziende; da quel momento in poi sarà permesso alle due aziende di creare *regole di inoltro* l'una verso l'altra (per il momento, è consentita la gestione di accordi bilaterali; vale a dire, se un'azienda A invia una *partnership request* a B e se l'azienda B dovesse accettarla, entrambe potranno definire reciproche *regole di inoltro*).

Poiché con l'introduzione delle *partnership requests* non è più possibile che una qualsiasi azienda iscritta su iChain possa a priori definire una *regola di inoltro* nei

confronti di una seconda azienda senza che questa ne abbia prima accettato una richiesta di collaborazione dalla prima, un altro problema viene così automaticamente risolto: non è più possibile copiare dei dati da un database a un altro senza che il proprietario del database su cui viene eseguita la copia ne sia consapevole e non è, di conseguenza, possibile che esso riceva dei dati che non desidera avere. Tale situazione può apparire paradossale se si considera quanto detto finora sul concetto di "segreto industriale" particolarmente caro alle aziende e una domanda sorge dunque spontanea: perché qualcuno dovrebbe volontariamente inviare a terzi i propri dati, che magari costituiscono appunto un segreto industriale, senza che sia stato prima definito un accordo? A tale domanda, certamente lecita, si forniscono tre possibili risposte:

1. trasmissione errata involontaria: un utente che gestisce l'account di un'azienda A registrata su iChain potrebbe confondersi nel vedersi comparire un elenco, potenzialmente lungo, di aziende e potrebbe erroneamente definire una *regola di inoltro* verso un'azienda con cui non voleva realmente condividere informazioni su certi dati;

2. trasmissione consapevole disonesta: un utente che gestisce l'account di un'azienda A registrata su iChain, corrotto da terzi potrebbe illecitamente trasmettere ad altri dei dati riservati dell'azienda per cui lavora;

3. attacco informatico (un problema per cui bisognerà sempre pensare a dei requisiti e a dei sistemi di sicurezza continuamente aggiornati): un pirata informatico potrebbe riuscire ad accedere all'account di un dipendente di un'azienda A (perché è riuscito, per esempio, a sottrargli con qualche sotterfugio le credenziali di accesso) e trasmettere dati di tale azienda a terzi per una ragione qualsiasi, impredicibile; per esempio, con lo scopo di riempire il database di un'azienda B, reale target dell'hacker, con dati "spazzatura" al fine di saturarne le risorse (attacco *DoS*).

A seguire si discuterà in merito alla progettazione e alla implementazione delle *partnership requests*.

3.5.1 Architettura delle richieste di partnership

Quella delle richieste di partnership è una caratteristica aggiunta di sana pianta alla piattaforma iChain e ha quindi permesso la massima libertà di scelta a livello progettuale/architetturale. Si è scelto di adottare il pattern *Controller-Service-Repository*[16].
In figura 3.3 è possibile visualizzare l'organizzazione in strati di tale pattern.

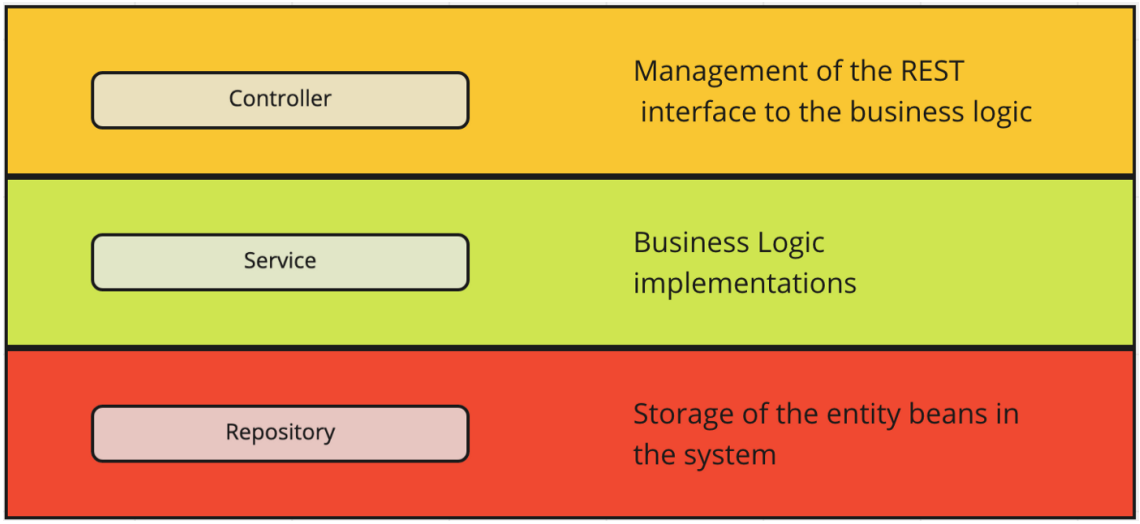

Figura 3.3: una rappresentazione del pattern Controller-Service-Repository [17].

Seguendo tale pattern, sono dunque stati definiti i tre seguenti layers:

1. *Partnership Request Controller*, che contiene e gestisce tutti gli endpoints REST che la piattaforma iChain espone ai client e fa da tramite per coloro che vogliono accedere alla logica di business relativa alle richieste di partnership;

2. *Partnership Request Service*, che implementa effettivamente tutta la logica di business che concerne le richieste di partnership;

3. *Partnership Request Repository*, che si occupa di immagazzinare nel database e recuperare da esso tutte le informazioni necessarie relative alle richieste di partnership.

La scelta è ricaduta su tale pattern sia per una questione di gusto personale sia, soprattutto, per l'ottimo livello di separazione dei concetti che tale pattern offre e che permette così di avere del codice maggiormente leggibile, mantenibile e testabile.

3.5.2 Modellazione delle richieste di partnership

La definizione del data model di una *partnership request* non ha richiesto particolare attenzione, in quanto tutti i campi che lo compongono sono stati individuati senza particolari difficoltà. Essi sono i seguenti:

- *id*, usato per identificare univocamente la richiesta di partnership all'interno della piattaforma;
- *brand_ domain1*, che rappresenta il dominio dell'azienda che ha inviato una richiesta di partnership a una seconda azienda;
- *brand_ domain2*, che rappresenta il dominio dell'azienda ricevente la richiesta di partnership e a cui spetta la scelta di decidere se accettarla o meno;
- *status*, che contiene lo stato della richiesta di partnership e che, al momento, può assumere uno dei seguenti valori:
  1. *pending*, che indica che la richiesta di partnership è in attesa di essere accettata dalla azienda destinataria. È il valore che la richiesta di partnership possiede quando viene generata;
  2. *rejected*, che indica che la richiesta di partnership è stata rifiutata dalla azienda destinataria;
  3. *active*, che indica che la richiesta di partnership è stata accettata dalla azienda destinataria. A partire da questo momento, le due aziende coinvolte, il cui dominio è contenuto nei campi *brand_ domain1* e *brand_ domain2*, potranno definire reciprocamente delle *regole di inoltro*. Si fa notare che attualmente tale stato è da considerare come se fosse *accepted* (accettata) ma si è usato fin da subito il valore *active* (attiva) per consentire, in una versione futura di iChain, l'eventuale possibilità di interrompere una collaborazione tra due aziende. Nel qual caso occorrerebbe un nuovo valore per il campo *status*;
- request_time, che contiene la data e l'ora in cui la richiesta di partnership è stata generata e inviata;

- *expire_time*, che contiene la data e l'ora entro cui la richiesta di partnership potrà essere accettata. Oltre tale limite temporale, la piattaforma considererà automaticamente respinta la richiesta;

- *token_to_accept*, che contiene il token JWT generato dalla piattaforma e inviato tramite email al destinatario della richiesta di partnership. Quest'ultimo potrà utilizzare il token entro e non oltre il momento temporale definito dal campo *expire_time* per accettare, se lo desidera, la richiesta di partnership ricevuta.

3.5.3 Il Partnership Request Service

Il Partnership Request Service è il layer che, come si è precedentemente detto, implementa la logica di business inerente le *partnership requests* e quindi svolge tutte le operazioni atte a generare, accettare e rifiutare una richiesta di partnership. In questo paragrafo si descriverà sommariamente la logica di business implementata dal *Partnership Request Service*, limitandosi a descrivere la sequenza di passi svolti per ciascuna operazione senza soffermarsi più di tanto, quindi, sui particolari del codice.
La prima operazione descritta sarà la generazione di una richiesta di partnership effettuata da un'azienda A a un'azienda B. Il *Partnership Request Service* svolge i seguenti passi:

1. verifica la validità dei domini aziendali, che devono entrambi esistere su iChain e non devono appartenere allo stesso utente. In caso negativo, non eseguirà altro;

2. verifica che non ci sia né una richiesta attiva né una richiesta pendente tra i due domini. In caso negativo, non eseguirà altro;

3. se entrambe le verifiche sono soddisfatte, memorizza nel database la richiesta di partnership, riempiendo opportunamente i vari campi del data model associato;

4. crea un token JWT che consente, a chi è autorizzato a farlo (vale a dire il dominio aziendale destinatario della richiesta), di accettare la richiesta

di partnership. Ogni token JWT generato viene riempito con le seguenti informazioni:

(a) l'identificativo univoco della richiesta di partnership;

(b) il dominio aziendale destinatario;

(c) lo username dell'utente autorizzato (cioè colui che ha accesso al dominio aziendale destinatario) ad usare il token JWT per accettare la richiesta di partnership;

(d) la scadenza del token, ovvero il limite temporale ultimo entro cui si potrà accettare la richiesta di partnership. Tale scadenza contiene lo stesso valore che è salvato nel campo *expire_time* del data model associato a tale richiesta di partnership e di cui si è detto al punto precedente;

5. invia una email di notifica al destinatario della richiesta di partnership. Tale email includerà anche il token JWT da usare per accettare, se interessati, la richiesta.

La seconda operazione descritta sarà l'accettazione, da parte di un'azienda B, di una richiesta di partnership avanzata da un'azienda A. Il *Partnership Request Service* svolge i seguenti passi:

1. valida l'autenticità del token JWT ricevuto, verifica che non sia scaduto e verifica che chi lo abbia inviato sia autorizzato a usarlo. In caso negativo, non eseguirà altro;

2. se le verifiche precedenti sono soddisfatte, verifica che la richiesta di partnership non sia già stata accettata. In caso negativo, non eseguirà altro;

3. se la richiesta di partnership non è stata ancora accettata (cioè se è ancora in uno stato pendente), contrassegnerà tale richiesta come attiva e, come si è detto, ciò abiliterà i due domini aziendali coinvolti a creare *regole di inoltro* fra di loro.

L'ultima operazione descritta sarà, ovviamente, il rifiuto di una richiesta di partnership. Se un dominio aziendale B non è interessato ad accettare la richiesta di

partnership di un dominio aziendale A, non deve svolgere alcuna operazione. La piattaforma iChain, o per meglio dire il *Partnership Request Service*, la considererà automaticamente come rifiutata una volta che essa sarà scaduta. Se ci si è scordati di accettare la richiesta ma essa è stata già respinta automaticamente, le due aziende dovranno mettersi nuovamente in contatto e una delle due dovrà generare una nuova richiesta di partnership verso l'altra, ripartendo così dalla prima operazione descritta.

A livello implementativo, per gestire il rifiuto automatico delle richieste si è utilizzata la libreria *timeloop*[18], una libreria python che consente di eseguire periodicamente un blocco di codice in maniera del tutto automatica ancora una volta usando semplicemente un'annotazione.

Di seguito si mostra il blocco di codice che usa, appunto, la libreria *timeloop*, incaricato di controllare periodicamente se ci sono richieste di partnership scadute e, in caso, contrassegnarle come respinte.

```
@partnership_request_service_timeloop.job(interval=datetime.timedelta(seconds=60))
def reject_expired_pending_partnership_requests():
    partnership_request_service.partnership_request_repository.reject_expired_pending_partnership_requests()

# Tale metodo si trova nel Partnership Request Repository
def reject_expired_pending_partnership_requests(self) -> int:
    now = datetime.now()
    query_filter = {
        "expire_time": {"$lt": now},
        "status": PartnershipRequestStatusEnum.PENDING
    }
    update_statement = {
        "$set": {"status": PartnershipRequestStatusEnum.REJECTED}
    }
    update_result = self._coll.update_many(query_filter, update_statement)
    return update_result.modified_count
```

3.5.4 Il Partnership Request Controller

Il *Partnership Request Controller*, come si è visto, è il layer incaricato di fornire una serie di endpoints ai clients e fa da tramite con *Partnership Request Service*.
Gli endpoints REST esposti sono:

- l'endpoint POST api/partnership_requests, che consente di creare una richiesta di partnership. Richiede in input un insieme (non vuoto) dei domini a cui fare una richiesta di partnership. Il mittente di tale richiesta sarà automaticamente desunto guardando il dominio aziendale dell'utente che ha contattato tale endpoint;

- l'endpoint POST api/partnership_requests/accept/token, che consente di accettare una richiesta di partnership. Richiede in input un token JWT che sarà poi, come si è visto, usato dal *Partnership Request Service* per accettare (se risultano soddisfatte le condizione di cui si è discusso in precedenza) la richiesta di partnership.

- l'endpoint GET api/partnership_requests/pending, che restituisce la lista di richieste di partnership pendenti fatte da chi ha contattato l'endpoint;

- l'endpoint GET api/partnership_requests/active, che restituisce la lista di richieste di partnership attive che coinvolgono il soggetto che ha contattato l'endpoint;

- l'endpoint GET api/partnership_requests/to_accept, che restituisce la lista di richieste di partnership che devono ancora essere accettate da chi ha contattato l'endpoint.

Oltre a introdurre i sopraelencati endpoints nel *Partnership Request Controller* è stato anche necessario modificare un ulteriore endpoint già esistente, che non è gestito dal textitPartnership Request Controller. L'endpoint in questione è *POST api/rules* che consente di creare una *regola di inoltro* tra due domini. La modifica fatta consiste nell'impedire che sia possibile creare una *regola di inoltro* tra due domini aziendali fra i quali non è ancora stata definita alcuna forma di collaborazione (cioè, non vi è una *partnership request* nello stato attivo).

Si conclude la trattazione delle *partnership request* con due figure: la prima, la figura 3.4, illustra la sezione su iChain *My Partners* in cui è possibile visualizzare l'elenco delle aziende con cui si ha una collaborazione attiva; la seconda, la figura 3.5, illustra la sezione su iChain in cui è possibile visualizzare l'elenco di tutte le aziende registrate su iChain. Attraverso questa seconda sezione è anche possibile visualizzare se per un data azienda si ha una collaborazione attiva, se si ha una richiesta di partnership pendente, se si ha una richiesta di partnership da accettare o è possibile inviare richieste di partnership.

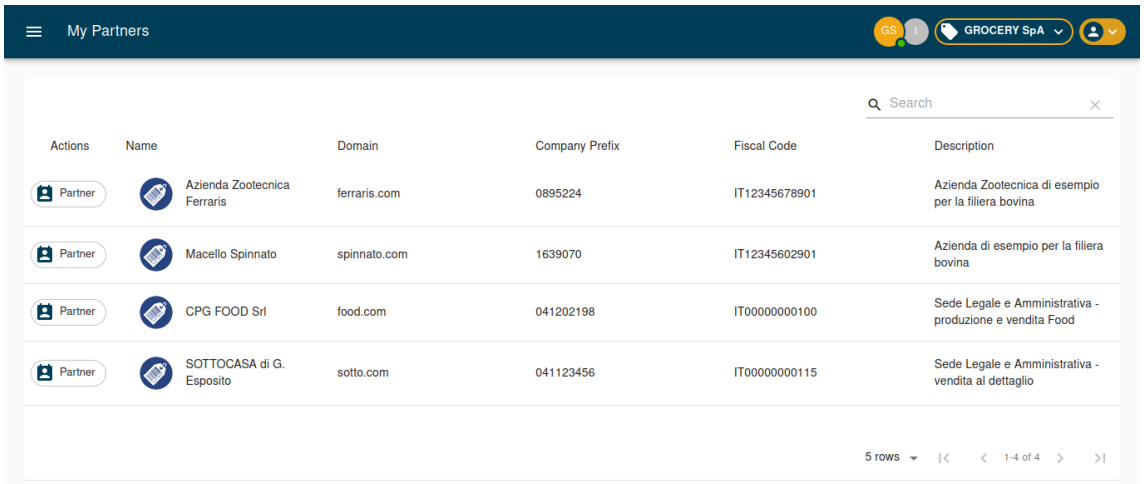

Figura 3.4: la sezione 'My Partners' su iChain.

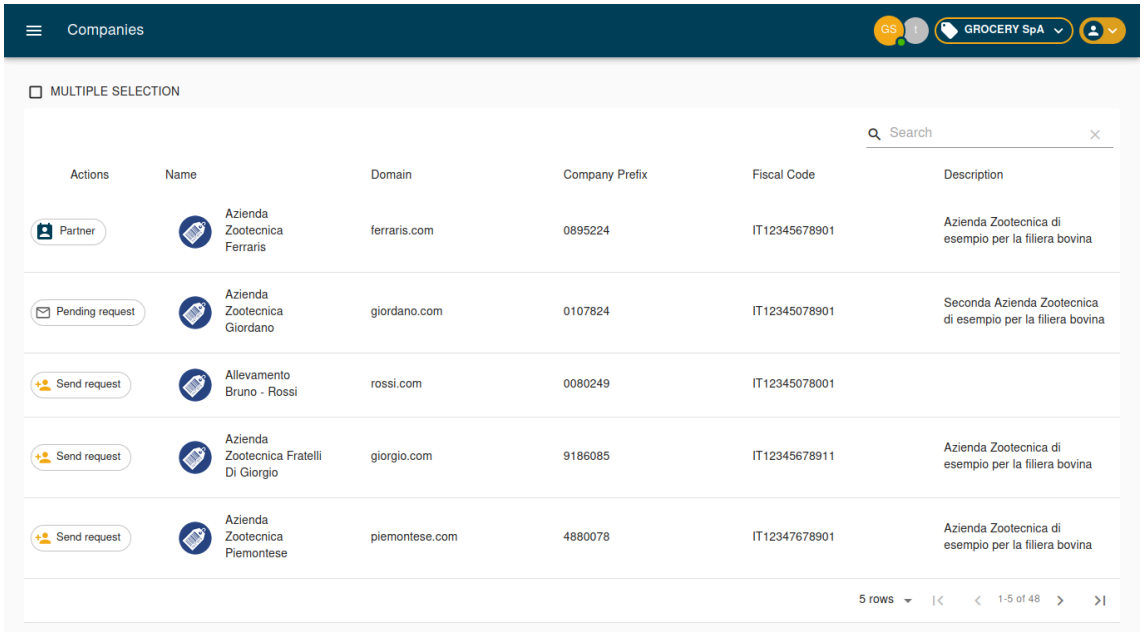

Figura 3.5: la sezione contenente tutte le aziende registrate su iChain.

4

Un ambiente di test

Il secondo obiettivo di questa tesi è stato quello di mettere a punto un ambiente di test che permetta di fare esperimenti e test sperimentali vari sui dati che si trovano su iChain senza che questi vengano in qualche modo compromessi o corrotti.
Come più volte rimarcato in precedenza, i due principali scopi della piattaforma iChain sono i seguenti:

1. consentire a una qualsiasi azienda di poter avere una visione di insieme (*helicopter view)* di quanto avviene in una *supply chain*, permettendole così di poter tracciare i propri prodotti in tempo reale e di visualizzare i flussi produttivi a livello di filiera e la storia di ciascun prodotto coinvolto in uno di essi;

2. stabilire delle regole di *trasparenza* verso terzi, ossia regole che definiscono cosa condividere con gli altri del proprio flusso produttivo interno affinché questi possano usufruirne per avere una visione del flusso produttivo a livello di filiera.

Tuttavia, utilizzare i dati raccolti sulla *trasparenza* e sulla *tracciabilità* a livello di filiera solamente per averne una visualizzazione grafica sarebbe, per quanto utile, molto limitante. Le aziende, infatti, spesso sono interessate ai *KPI* (Key Performance Indicator), in italiano ICP (Indicatore chiave di prestazione). I KPI costituiscono una serie di metriche che un'azienda usa per valutare le proprie prestazioni nel tempo, cioè per misurare i progressi fatti nel conseguimento dei propri obiettivi e per confrontarsi con altre aziende attive nel medesimo settore di produzione [19]. Per tale ragione iChain non si limita semplicemente a essere una piattaforma generica ma, su richiesta della singola azienda, offre anche la possibilità di introdurre dei *plugins* "privati", cioè degli strumenti di analisi a livello di filiera sui dati raccolti e che sono realizzati appositamente per rappresentare i KPI richiesti da tale azienda. Il dover realizzare plugins che variano da un'azienda all'altra, ciascuna avente i propri dati e con le proprie necessità, è il primo motivo per cui serve un ambiente

di test in cui poter verificare, in fase di sviluppo, il funzionamento di ogni plugin sui dati di test senza compromettere in alcuno modo i dati reali che l'azienda ha immesso su iChain. Solo una volta che si ha la certezza che tali plugins sono in grado di svolgere le analisi desiderate, essi saranno applicati sui dati concreti dell'azienda presenti sulla piattaforma iChain.
Il primo plugin realizzato su iChain - e che, di fatto, è fornito a tutti i clienti - consente a un'azienda di inoltrare i propri eventi EPCIS ai partner con cui si sono definite una o più *regole di inoltro* e di costruire, sfruttando sia gli eventi EPCIS prodotti internamente sia quelli che sono stati condivisi da terzi, il flusso produttivo a livello di filiera e la storia di ogni prodotto presente in tale flusso. Tale costruzione è svolta dai due algoritmi principali della piattaforma già menzionati nel capitolo precedente, vale a dire il *history diagram algoritm* e il *marble diagram algoritm*. Ogni modifica a questi due algoritmi, che sono di loro intrinsecamente complessi, potrebbe compromettere il funzionamento della piattaforma; ecco, dunque, una seconda ragione per cui è richiesta la presenza di un ambiente di test. Inoltre, come si è visto, con l'introduzione della nuova tipologia di evento EPCIS su iChain, l'AssociationEvent, è stato necessario modificare i due algoritmi in questione rendendo, di fatto, l'opportunità di avere un ambiente di test una necessità concreta e immediata. Come si vedrà meglio nel corso di questo capitolo, l'elemento chiave che ha permesso di creare l'ambiente di test desiderato è stata la piattaforma *Debezium*. Tuttavia, prima di discutere cosa essa sia, cosa offra e come ha permesso di costruire un ambiente di test è opportuno esaminare brevemente quella che era l'architettura di iChain prima di questa tesi. Il presente capitolo sarà, dunque, strutturato nel modo seguente:

- nel primo paragrafo si vedrà sommariamente qual era la struttura architetturale di iChain; dopodiché, si discuterà del perché tale struttura non fosse sufficiente per costruire un ambiente di test;
- nel secondo paragrafo si illustreranno le caratteristiche della piattaforma *Debezium*;
- nel terzo paragrafo si vedrà come è stato messo a punto l'ambiente di test desiderato;

- nel quarto paragrafo si discuteranno alcune piccole migliorie che sono state adottate per agevolare i test;
- nel quinto e ultimo paragrafo si menzioneranno brevemente alcuni problemi che, grazie all'uso di un ambiente di test, sono stati riscontrati sulla piattaforma iChain.

4.1 L'architettura originaria di iChain

La struttura portante su cui si fonda iChain è costituita dai tre seguenti elementi [2]:

1. *Apache Kafka* [20] o più semplicemente *Kafka*;
2. *MongoDB* [21];
3. *Faust* [22].

Ciascuna delle tre tecnologie sarà discussa in un paragrafo dedicato.

4.1.1 Apache Kafka

Kafka è una piattaforma distribuita per il data streaming che consente di pubblicare, sottoscrivere, archiviare ed elaborare flussi di dati in tempo reale. Tali flussi di dati possono provenire da più sorgenti (dette *data-producer*) e possono essere distribuiti a più riceventi (detti *data-consumer*) [23].
Uno degli elementi principali di *kafka* è il *topic*. Un *topic* è il luogo (luogo inteso in termini astratti) in cui un certo flusso di dati è pubblicato, preservando l'ordine di arrivo di ciascun messaggio; un *topic* è diviso in un certo numero di partizioni (almeno una) [24]. Ogni messaggio pubblicato su un *topic* ha una chiave, un valore (entrambi possono essere delle stringhe di caratteri o una sequenza di byte) e un timestamp (che è usato per stabilire l'ordine di arrivo dei vari messaggi). La chiave è usata per scegliere la partizione del *topic* (se ce n'è più di una) in cui pubblicare il messaggio (i messaggi aventi la stessa chiave sono pubblicati nella stessa partizione).
Un secondo elemento di rilievo di *kafka* è il *kafka connector*. I *kafka connectors* sono

dei componenti che permettono di importare dati da sistemi esterni (e.g. Elasticsearch, Hadoop o qualsiasi tipo di database) in un *topic kafka* o di esportare dati da un *topic kafka* a un sistema esterno [25]. Esistono due classificazioni di *kafka connectors*:

1. i *source connectors*, che catturano dei dati da un sistema esterno e li pubblicano in uno o più *kafka topics*;
2. i *sink connectors*, che consumano dei messaggi da uno o più *kafka topics* e li forniscono a un sistema esterno.

Un ulteriore elemento di rilievo di *kafka* è il *kafka broker*. I *kafka brokers* sono responsabili della gestione dei *kafka topics* e delle relative partizioni. *Kafka* appare quindi come una tecnologia molto all'avanguardia. Alcuni dei suoi punti di forza sono:

- ha un elevato *throughput* con perdita nulla dei messaggi;
- garantisce l'ordine di arrivo dei messaggi;
- è scalabile orizzontalmente, quindi particolarmente adatto a quelle applicazioni che devono gestire sempre più messaggi nel tempo;
- è resistente ai guasti;
- è in grado di interagire con un'ampia gamma di database e di sistemi;
- ha una vasta comunità online;
- esistono diverse librerie per diversi linguaggi di programmazione che permettono di usare *kafka* all'interno delle applicazioni.

Kafka è quindi una tecnologia usata da migliaia di organizzazioni, tra cui i giganti del web.
Nello caso di iChain, *kafka* è usato nelle seguenti situazioni:

- a ogni azienda che si registra sulla piattaforma vengono assegnati dei *kafka topics*. In uno di essi sono pubblicati tutti i dati in ingresso per quell'azienda (per esempio gli eventi EPCIS generati da essa). I dati pubblicati sul *topic*

saranno poi esaminati da uno *stream processor* che, in base alle *regole di inoltro* che tale azienda ha definito, li trasmette (ovviamente se soddisfano almeno una *regola di inoltro*) alle aziende partner autorizzate a riceverli. La trasmissione a una azienda partner è fatta, a sua volta, pubblicando tale evento EPCIS su un *kafka topic* dell'azienda partner;

- i vari plugins "privati" di ogni azienda consumano i dati dai *kafka topic* al fine di eseguirne una qualche analisi di interesse;

- presiede il consumo di messaggi dai *topics* per poi memorizzare i dati ottenuti su un database (in particolare su una o più collezioni all'intero di *MongoDB*). Ciò è fatto tramite i *sink connectors*);

- presiede il rilevamento di particolari modifiche che avvengono su un database (in particolare su una o più collezioni all'intero di *MongoDB*) per poi pubblicare i relativi dati su un *kafka topic*. Ciò è fatto tramite i *source connectors*.

4.1.2 MongoDB

MongoDB è un database NoSQL orientato ai documenti. Ogni documento è in formato JSON-like e non ha necessariamente uno schema fisso [26].
MongoDB raggruppa i documenti in delle collezioni, che di fatto sono analoghe alle tabelle nei database relazionali. I punti di forza di *MongoDB* sono:

- la scalabilità orizzontale, che consente quindi di gestire un'ampia mole di dati senza intaccare più di tanto le prestazioni all'aumentare dei dati;

- è tollerante ai guasti in quanto consente di replicare facilmente i database;

- la flessibilità, data dal non avere uno schema rigido come invece si ha nei database relazionali (attenzione: non si vuole in alcun modo dire che i database NoSQL siano migliori dei database relazionali; la caratteristica di avere uno schema non rigido, infatti, rappresenta in alcuni casi uno svantaggio significativo).

Su iChain si è scelto di usare *MongoDB* per via del fatto che esso memorizza i documenti in un formato JSON-like ed è, quindi, altamente compatibile con la rappresentazione JSON degli eventi EPCIS (definita dalla versione 2.0 dello standard ma usata fin da subito su iChain, come si è visto) e degli altri dati gestiti dalla piattaforma.
Ogni azienda registrata su iChain possiede il proprio database privato, separato da quelli delle altre aziende, in cui sono contenuti i propri dati (e.g. gli eventi EPCIS) più i dati che ha ricevuto da terzi; in più vi è almeno un database dedicato alla piattaforma stessa. Ognuno dei database è collegato a *kafka* tramite i *kafka connectors* su menzionati: ogni nuovo evento EPCIS salvato in un database viene pubblicato su un *kafka topic* per essere poi elaborato da uno o più *stream processors*; inoltre, alcuni dati scritti su un topic possono essere consumati per poi essere immagazzinati su un database (per esempio, quando si legge dal proprio *topic* dedicato un evento EPCIS che è stato ricevuto da un'azienda partner tale evento EPCIS viene successivamente memorizzato sul proprio database).

4.1.3 Faust

Prima di descrivere in dettaglio la libreria *Faust*, è bene fare un brevissimo accenno su *Kafka streams*. *Kafka streams* è una libreria per scrivere applicazioni, o in linguaggio Java o in linguaggio SCALA, che consente di processare i messaggi memorizzati su dei *kafka topics* per poi svolgere, per ciascun messaggio, altre operazioni (per esempio, scrivere i dati risultanti su un altro o su altri *kafka topic*). [27]. L'attore che effettua il processamento dei dati è chiamato *stream processor* (i.e. elaboratore del flusso).
Ciò premesso, si può ora evidenziare che *Faust* è una libreria per l'elaborazione del flusso che porta l'idea della libreria *kafka streams* in python [28]. La libreria *Faust* consente di definire uno *stream processor* tramite una semplice annotazione; tali *stream processors* sono anche detti *agents*. Su iChain, ve ne sono diversi: uno di questi, per esempio, ha la funzione di verificare se, per ogni nuovo evento EPCIS memorizzato sul database di un'azienda, tale evento soddisfa una o più *regole di inoltro* e, in caso affermativo, lo invia ai partner autorizzati.

4.1.4 "Dockerizzazione" dell'architettura

I tre componenti dell'architettura di cui si è parlato, sono stati "dockerizzati" per poter essere facilmente caricati su un servizio cloud. Nella versione iniziale dell'architettura, le immagini docker usate erano le seguenti:

- un'immagine custom contenente la piattaforma iChain (i.e. che esegue sia il codice di back-end sia il codice di front-end);
- un'immagine custom per *faust*, cioè un'immagine su cui è eseguita un'applicazione contenente i vari *stream processors*;
- un'immagine che esegue un'istanza di *MongoDB*;
- l'immagine *bitnami/zookeeper*, che è usata per eseguire *Zookeeper*, cioè un servizio per gestire e coordinare i *kafka brokers* [29];
- un'immagine custom generata a partire dall'immagine *bitnami/kafka*, che è usata per eseguire *Apache Kafka* [30];
- l'immagine *obsidiandynamics/kafdrop*, che è un tool di amministrazione e di monitoraggio dei *kafka topics* [31];
- l'immagine *confluentinc/cp-kafka-rest*, che fornisce una serie di endpoints REST per interfacciarsi a *kakfa* [32] (per esempio, fornisce endpoints per creare/-cancellare un *kafka topic*).

4.1.5 Un'architettura inadeguata per un ambiente di test

L'architettura originaria di iChain non presenta particolari problemi ed è adatta anche per essere usata in un ambiente di produzione. Tuttavia, non è, però, adatta per ottenere l'ambiente di test flessibile e scalabile desiderato.
Tale ambiente di test, infatti, richiede che sia possibile creare al volo delle collezioni di dati su *MongoDB* di test, dei *kafka topics*, dei *kafka connectors*. Con l'architettura originaria è possibile ottenere le prime due cose, ma non si possono creare dei *kafka connectors* al volo; non sarebbe stato possibile, quindi, realizzare l'ambiente di test desiderato in quanto verrebbero "sporcati" i *kafka connectors* "ufficiali" che

avrebbero poi, a loro volta, corrotto i *kafka topics* e le collezioni *MongoDB* originali. Tale mancanza è stata risolta introducendo un ulteriore elemento all'architettura di iChain: *Debezium*. Tale tecnologia aggiuntiva sarà meglio descritta nel successivo paragrafo.

4.2
Un nuovo importante elemento per l'architettura di iChain: Debezium

Debezium è una piattaforma open-source, basata su *Apache Kafka*, in grado di rilevare i cambiamenti a uno o più database semplicemente leggendo il loro *transaction log* affinché le applicazioni possano poi essere informate di tali modifiche e reagire in qualche maniera più opportuna [33].
Debezium, quindi, è una piattaforma altamente progettata per la *CDC* (Change Data Capture). *CDC* è un pattern che permette di rilevare cambiamenti nei dati, al fine di poter intraprendere una qualche azione a partire da tali dati [34].
I vantaggi forniti da Debezium sono:

1. garanzia assoluta che qualsiasi cambiamento avvenga su un database (inserimento di dati, modifica di dati, cancellazione di dati) sia rilevato;

2. non richiede di dover cambiare i *data models* definiti in un'applicazione;

3. offre un vasto insieme di *connectors* per svariati database, incluso *MongoDB*. In più, permette di creare, modificare e cancellare un qualsiasi *connector* al volo semplicemente usando un opportuno endpoint REST.

In *Debezium*, i *connectors* pubblicano tutti gli eventi di cambiamenti su dei *kafka topics*.
Debezium risulta perciò essere una piattaforma estremamente utile per iChain, per le seguenti ragioni:

1. fa uso di *Kafka*, che è uno degli elementi cardine già presente nell'architettura di iChain;

2. fornisce *connectors* per *MongoDB*, che è appunto il tipo di database scelto per iChain;

3. consente di creare, modificare e cancellare i *connectors* in qualsiasi momento fornendo, di conseguenza, il tassello mancante per poter realizzare l'ambiente di test voluto.

La figura 4.1 illustra come si colloca *Debezium* all'interno di un'applicazione python che si interfaccia con *Kafka* e *MongoDB*.

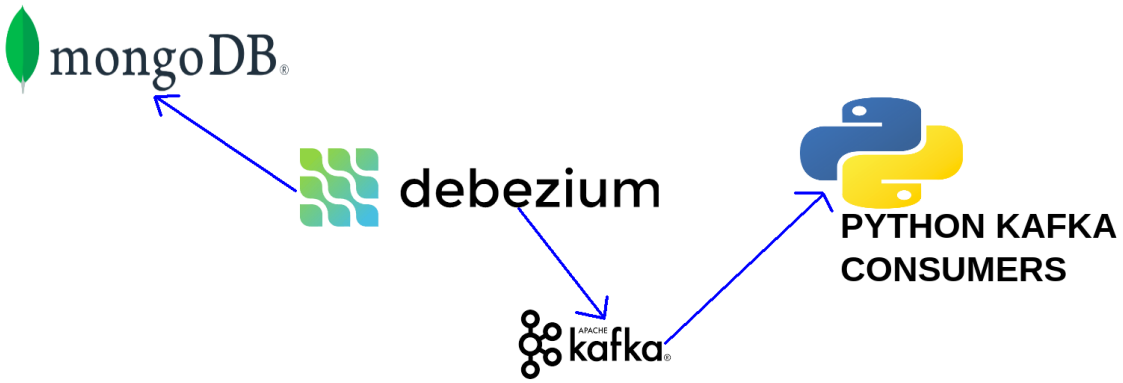

Figura 4.1: il collocamento di *Debezium* in un'applicazione python che usa *MongoDB* e *Kafka*.

Prima di discutere come è stato realizzato l'ambiente di test a partire da *Debezium*, nel prossimo paragrafo si fornirà una breve descrizione sui *kafka connector* per *MongoDB* offerti da *Debezium*. Ulteriori dettagli, anche con un maggior livello di approfondimento, sono ovviamente disponibili consultando la documentazione ufficiale di *Debezium*.

4.2.1 I connectors di Debezium per MongoDB

Un *connector* per *MongoDB* di *Debezium* osserva un *replica set* (cioè un insieme di server che hanno tutti copie degli stessi dati) di *MongoDB* per rilevare dei cambiamenti in dei database o in specifiche collezioni di un certo database. I cambiamenti rilevati sono poi registrati come messaggi su almeno un *kafka topic* [35].
I *connectors* per *MongoDB* di *Debezium* hanno un numeroso insieme di proprietà di configurazione che possono essere settate per plasmare il *connector* più adatto per gli scopi di una specifica operazione. La maggior parte di queste proprietà di configurazione ha dei valori di default. Nella tabella 4.1 si elenca una serie di proprietà con relativa descrizione. Non saranno elencate tutte le proprietà di configurazione

di un *connector* ma sono quelle a cui, nell'ambito di questa tesi, sono stati assegnati dei valori non predefiniti.

Proprietà	**Descrizione**
name	Il nome logico del *connector*. Deve essere univoco. È usato anche come prefisso per i *kafka topics*.
connector.class	Il nome della classe Java corrispondente al *connector*. Per un *MongoDB connector* bisogna sempre indicare il valore *io.debezium.connector.mongodb.MongoDbConnector*.
mongodb.name	Il nome che identifica il *replica set* di MongoDB che il *connector* monitora.
mongodb.hosts	L'indirizzo del server su cui *MongoDB* è eseguito.
database.include.list	L'elenco dei database da monitorare. Se tale proprietà è settata, tutti i database non inclusi in tale lista non saranno monitorati; in caso contrario, tutti i database saranno monitorati di default.
collection.include.list	L'elenco delle collezioni da monitorare. Ogni collezione non inclusa in tale lista non sarà monitorata. Ogni elemento della lista deve avere la forma *databaseName.collectionName*.
database.history.kafka. bootstrap.servers	L'indirizzo del server su cui è eseguito *kafka*.

Tabella 4.1: alcune delle proprietà di configurazione per i *connectors* per *MongoDB* di *Debezium* [35].

4.2.2 Il Data Change Event

Un *connector* di *Debezium* genera un *DCE* (Data Change Event) per ogni operazione, a livello di documento, che inserisce, aggiorna o cancella dei dati. Ogni DCE ha una chiave e un valore in formato JSON [35].
La chiave di un DCE è solitamente, semplificandone di molto l'effettiva struttura, l'id del documento modificato.
Il valore di un DCE ha un payload i cui campi di maggiore interesse, per lo sviluppo di un ambiente di test su iChain, sono:

- il campo *op*, che è una stringa che indica l'operazione che ha causato l'evento. Può assumere uno dei seguenti quattro valori:

  1. *c*, che indica che è stato creato un nuovo documento all'interno della collezione monitorata dal *connector*;
  2. *u*, che indica che è stato modificato un documento già esistente all'interno della collezione monitorata dal *connector*;
  3. *d*, che indica che è stato cancellato un documento presente nella collezione monitorata dal *connector*;

4. *r*, che indica che è stato eseguito uno *snapshot*. Uno *snapshot* si verifica non appena un nuovo server si aggiunge a un *replica set* ed effettuerà subito una copia di tutti gli eventi passati che si erano verificati nel database.

- il campo *after*, che specifica lo stato del documento dopo che l'operazione è stata eseguita. In altri termini, contiene il documento stesso (il documento creato o il documento modificato). Naturalmente, se il documento viene cancellato tale campo non ha ragione di esistere.

4.3
La messa a punto di un ambiente di test

In considerazione di tutto quanto descritto e illustrato, è ora possibile esaminare i passi che hanno permesso la realizzazione dell'ambiente di test desiderato. Essi sono stati:

1. la definizione su *Docker* di una nuova architettura per la piattaforma iChain che rimpiazza parzialmente la precedente;

2. la parsificazione (parsing) del *Data Change Event* (*DCE*);

3. la messa a punto di un meccanismo atto a consentire la creazione dei *connectors* di test, dei *kafka topics* di test e delle collezioni di test.

4.3.1 La nuova architettura "dockerizzata"

Conseguentemente a tutto quanto si è discusso fin qui, il primo passo passo per la realizzazione di un ambiente di test è stata l'aggiunta di *Debezium* all'architettura di iChain, il vero elemento chiave che ha permesso l'attuazione di tale lavoro.
A tal riguardo sono state, quindi, modificate alcune delle immagini docker usate e menzionate nel paragrafo 4.1.4. Le immagini aggiunte sono:

- *debezium/zookeeper* [36] che, sebbene svolga le stesse funzioni della precedente immagine *bitnami/zookeeper*, si è resa comunque necessaria per eseguire la piattaforma *Debezium*;

- *debezium/kafka* [37] che, nonostante svolga le stesse funzioni della precedente immagine *bitnami/kafka*, si è resa anche essa necessaria per eseguire la piattaforma *Debezium*;

- l'immagine *debezium/connect* che gestisce tutti i tipi di *connectors* e fornisce parte della piattaforma *Debezium*. Tale immagine offre una serie di endpoints REST per la gestione dei *connectors* [38] ed è completamente nuova all'interno dell'architettura di iChain.

Alcuni degli endpoints REST che l'immagine *debezium/connect* fornisce per la gestione dei *connectors* sono [39]:

- l'endpoint GET /connectors/, che restituisce la lista dei *connectors* attivi;

- l'endpoint POST /connectors/, che consente di creare un *connector* inserendo (in formato JSON), nel corpo del payload, il nome che si vuole assegnare al *connector* e le varie proprietà di configurazione necessarie a creare il *connector* desiderato (alcune di esse, per esempio, sono state viste nella tabella 4.1);

- l'endpoint GET /connectors/<connector_name>/, che restituisce le informazioni sul *connector* avente il nome indicato (se tale *connector* esiste, ovviamente);

- l'endpoint GET /connectors/<connector_name>/config/, che restituisce le proprietà di configurazione del *connector* avente il nome indicato (se tale *connector* esiste, ovviamente);

- l'endpoint PUT /connectors/<connector_name>/config/, che consente di modificare le proprietà di configurazione del *connector* avente il nome indicato (se tale *connector* esiste, ovviamente);

- l'endpoint DELETE /connectors/<connector_name>/, che consente di cancellare il *connector* avente il il nome indicato (se tale *connector* esiste, ovviamente).

La presenza dei summenzionati endpoints REST ha costituito un ruolo assolutamente decisivo per la realizzazione di un ambiente di test in quanto consentono,

come si è visto, di gestire i *connectors* in tempo reale. Su iChain, infatti, la vecchia immagine docker usata per *Kafka* era stata leggermente modificata per consentire la creazione una tantum, in fase di inizializzazione di *Kafka*, di due *source connectors*: il primo per rilevare le modifiche alla collezione *MongoDB* contenente le varie *regole di inoltro*, il secondo per rilevare le modifiche alla collezione *MongoDB* contenente gli eventi EPCIS generati o ricevuti da partner. Per qualunque modifica ai *connectors* (i.e. per aggiungerne di nuovi, cancellare o modificare quelli già esistenti) sarebbe stato necessario cancellare l'immagine docker usata per *Kafka* e ricrearla; ciò, oltre a rendere impossibile la realizzazione di un ambiente di test, avrebbe potenzialmente potuto creare dei seri problemi anche in fase di produzione della piattaforma iChain, qualora si fosse avuta l'esigenza di creare un qualche *connector* per un qualsiasi scopo. Invece, grazie agli endpoints REST di cui si è parlato sopra, un qualsiasi *connector* può essere creato, cancellato o modificato al volo direttamente in fase di esecuzione; per tale ragione, non viene più creato alcun *connector* in fase di inizializzazione in quanto vengono ormai tutti creati direttamente in fase di esecuzione.

4.3.2 La parsificazione del Data Change Event

Come si è visto nel paragrafo 4.2.2, il DCE è generato da un *connector* quando questo rileva una qualche modifica a ciò che sta monitorando. È quindi importante estrarre da esso i campi di interesse che, per gli scopi di iChain, al momento sono i due campi del payload *after* e *op*.

A tal proposito è stata implementata una funzione di parsificazione, di cui si illustra di seguito il codice.

```
1 def parse_connector_dce(data_change_event: bytearray) -> Tuple[
    MongoOperationType, Optional[str], Optional[dict]]:
2     op_type: MongoOperationType = MongoOperationType.NONE
3     document_id: Optional[str] = None
4     after_document_json: Optional[dict] = None
5     dce_json = json.loads(data_change_event.decode('utf8').replace(
          "'", '"'))
6     payload = dce_json['payload']
7     op_type = get_op_type(payload['op'])
```

```
8     if op_type.is_insertion() or op_type == MongoOperationType.
          UPDATE:
9         after_document_json = json.loads(payload['after'])
10        document_id = after_document_json['_id']['$oid']
11    return op_type, document_id, after_document_json
```

Come si può facilmente vedere, questa funzione, denominata *parse_ connector_ dce*, a partire da una sequenza di byte costituenti il DCE restituisce in output tre elementi: il tipo di operazione svolta, l'id del documento coinvolto nell'operazione e lo stato del documento in seguito all'operazione svolta. I passi che tale funzione svolge sono i seguenti:

1. converte il DCE da una pura sequenza di byte a un JSON, al fine di renderne più agevole la manipolazione;

2. estrae il payload dal DCE;

3. dal payload di cui sopra estrae l'operazione svolta, leggendo il valore del campo *op*;

4. nel caso in cui l'operazione svolta sia o la creazione di un documento oppure la modifica di un documento, dal payload estrae l'id del documento e lo stato del documento in seguito all'operazione. Si ricorda che, in seguito a una cancellazione, un documento non ha più uno stato; sarebbe comunque stato possibile recuperare l'id del documento cancellato ma, al momento, su iChain la cancellazione non è un'operazione di interesse in quanto sulle collezioni attualmente monitorate dai *connectors* le cancellazione non devono, dal punto di vista logico, avvenire. Tanto per fare un esempio, una delle collezioni monitorate è quella contenente gli eventi EPCIS che un'azienda ha generato o ricevuto da terzi: è chiaro che tali eventi EPCIS, una volta generati, non possono essere cancellati o fatti sparire nel nulla.

Su iChain, la funzione di parsificazione del DCE è, ora, usata da diversi *stream processors* (o *agents*, secondo la terminologia di *faust*). Di per sé, tale funzione non è necessaria per avere un ambiente di test: ciò che serve a tale scopo, infatti, è

la piattaforma *Debezium*. Tuttavia, poiché i *connectors* forniti da *Debezium*, indipendentemente che siano o meno *connectors* di test, come si è detto restituiscono le informazioni sotto forma di DCE, è essenziale che gli *stream processors* siano in grado di estrapolare tali informazioni per poi poterle elaborare secondo la logica più opportuna per un certo scopo. Dunque, i vari *agents* useranno le informazioni estratte dal DCE (i.e. il tipo di operazione svolta, l'id del documento coinvolto nell'operazione e lo stato del documento in seguito all'operazione svolta) per eseguire le operazioni per cui sono stati programmati.

4.3.3 Creazione dei connectors, dei topic e delle collezioni

Dopo aver introdotto *Debezium*, all'interno del codice sia del back-end di iChain sia dei *plugins*, è stata definita una variabile di tipo stringa che funge da prefisso ai nomi dei *connectors*, dei *kafka topics* e delle collezione su *MongoDB*. In altri termini, modificando il contenuto di tale prefisso, la piattaforma iChain e gli *agents* faranno riferimento e useranno i *connectors*, i *kafka topics* e le collezioni con quel prefisso.
L'uso del prefisso consente agevolmente di avviare la piattaforma o gli *agents* in un ambiente di test. La convenzione usata è quella di avere una stringa non vuota come prefisso solo nel caso di ambienti di test. L'uso del prefisso consente pertanto di usare dei *connectors*, dei *kafka topics* e delle collezioni su MongoDB senza compromettere il contenuto degli stessi elementi nella versione in produzione della piattaforma e dei plugins.
L'uso di un prefisso fornisce anche due interessanti vantaggi, ovvero:

1. la possibilità di poter eseguire più ambienti di test in parallelo senza che essi possano in alcun modo interferire tra di loro (ciò a patto, naturalmente, che ogni ambiente di test in esecuzione utilizzi un prefisso differente dagli altri ambienti di test in esecuzione; in caso contrario, il codice eseguito deve essere protetto con dei meccanismi di sincronizzazione opportuni);

2. la possibilità di poter cambiare agevolmente (in quanto è sufficiente modificare il prefisso) anche in fase di esecuzione (se dovesse risultare necessario per qualche scopo, ovviamente) la *configurazione*, vale a dire *connectors*, *kafka topics* e collezioni usati.

L'uso del prefisso, tuttavia, non sarebbe stato da solo sufficiente né per realizzare un ambiente di test né per passare da una configurazione a un'altra in fase di esecuzione. Infatti, senza l'introduzione di *Debezium*, gli unici *connectors* utilizzabili erano quelli creati in fase di inizializzazione dell'immagine per *Kafka* su docker: dunque il cambio del prefisso, che si traduce nel cambio dei *connectors*, dei *kafka topics* e delle collezioni da usare - e che nel caso dei *kafka topics* e delle collezioni su *MongoDB* avrebbe funzionato, perché essi sono creati in automatico se non esistono - nel caso dei *connectors* avrebbe invece generato l'errore di *connectors* inesistenti con la conseguente impossibilità a rilevare un qualsiasi cambiamento alle collezioni da monitorare. Con l'introduzione e l'uso di *Debezium*, al contrario, il problema non si pone poiché, come si è visto, è possibile creare al volo dei *connectors* semplicemente invocando l'endpoint REST *POST /connectors* precedentemente menzionato.

Per passare in fase di esecuzione da una configurazione a un'altra, l'endpoint REST suddetto va invocato manualmente per creare i *connectors* necessari, se questi non esistono (al momento non è noto alcun caso per cui su iChain si debba cambiare la *configurazione* usata in fase di esecuzione, e pertanto non è stato ancora necessario progettare alcun meccanismo per automatizzare l'operazione).

Quando si avvia un ambiente di test, invece, i *connectors* necessari (che al momento sono solamente due, come si è visto nel paragrafo 4.3.1), se non esistono, vengono creati automaticamente.

Di seguito si riporta la funzione che, all'avvio dell'applicazione faust su cui sono eseguiti gli *stream processors*, crea i due *source connectors* necessari qualora essi non esistano già.

```
1 def try_create_connectors():
2     PREFIX: str = "SOURCE_"
3     POSTFIX: str = ""
4     DB_NAME: str = settings.MONGO_DB
5     EPCIS_EVENT_COLL = MongoCollectionsEnumClass.
          CAPTURE_EPCIS_EVENTS
6     API_RULES_COLL = MongoCollectionsEnumClass.API_RULES
7     CONNECTORS_TO_CREATE: dict = {
8         PREFIX + EPCIS_EVENT_COLL + POSTFIX: {
9             "name": PREFIX + EPCIS_EVENT_COLL + POSTFIX,
```

```
10            "config": {
11                "connector.class": "io.debezium.connector.mongodb.
                      MongoDbConnector",
12                "tasks.max": "1",
13                "mongodb.name": settings.MONGO_SERVICE,
14                "mongodb.hosts": settings.MONGO_SERVICE + ':' +
                      settings.MONGO_PORT,
15                "database.include.list": DB_NAME,
16                "collection.include.list": DB_NAME + '.' +
                      EPCIS_EVENT_COLL,
17                "database.history.kafka.bootstrap.servers":
                      settings.KAFKA_SERVICE + ':' + settings.
                      KAFKA_PORT
18            }
19        },
20        PREFIX + API_RULES_COLL + POSTFIX: {
21            "name": PREFIX + API_RULES_COLL + POSTFIX,
22            "config": {
23                "connector.class": "io.debezium.connector.mongodb.
                      MongoDbConnector",
24                "tasks.max": "1",
25                "mongodb.name": settings.MONGO_SERVICE,
26                "mongodb.hosts": settings.MONGO_SERVICE + ':' +
                      settings.MONGO_PORT,
27                "database.include.list": DB_NAME,
28                "collection.include.list": DB_NAME + '.' +
                      API_RULES_COLL,
29                "database.history.kafka.bootstrap.servers":
                      settings.KAFKA_SERVICE + ':' + settings.
                      KAFKA_PORT
30            }
31        }
32    }
33    # Derive from connectors info a set containing connectors names
34    CONNECTORS_TO_CREATE_NAMES: set = set()
35    for key in CONNECTORS_TO_CREATE:
36        CONNECTORS_TO_CREATE_NAMES.add(key)
37
```

```
38      try:
39          LINFO("Getting list of existing connectors")
40          connectors_set = get_connectors_set(settings.DEBEZIUM_HOST)
41          LINFO("Successfully received list of existing connectors")
42
43          # Getting set of connectors to create making the set
               difference between the connectors that I would have
44          # and the connectors name that already exist.
45          missing_connectors_names: set = CONNECTORS_TO_CREATE_NAMES.
               difference(connectors_set)
46          if len(missing_connectors_names) >= 1:
47              # Try to create each missing source connector...
48              print(f"The following source connectors miss: {
                   missing_connectors_names}")
49              for connector_name in missing_connectors_names:
50                  print(f"Source connector {connector_name} is being
                       created")
51                  payload_to_send = CONNECTORS_TO_CREATE[
                       connector_name]
52                  create_connector(settings.DEBEZIUM_HOST,
                       payload_to_send)
53                  print(f"Source connector {connector_name} was
                       created")
54          else:
55              print("All needed source connectors already exist")
56      except Exception as exception:
57          exit(1)
```

Tale funzione, chiamata try_create_connectors, come prima cosa specifica quali *connectors* andrebbero creati definendone i nomi e le proprietà di configurazione. Dopodiché verifica quali di questi *connectors* già esistono e crea solo quelli mancanti.

Si ritiene estremamente importante precisare come il prefisso usato all'interno di questa funzione, ossia la costante *PREFIX*, non sia lo stesso prefisso menzionato in precedenza. Quest'ultimo, infatti, si propaga nel nome del *connector* indirettamente, in quanto tale nome è della forma *PREFIX* + *NOME_COLLEZIONE_DA_MONITORARE* + *POSTFIX* e il nome della collezione contiene già il prefisso di cui si è

parlato all'inizio di questo paragrafo.

La funzione try_create_connectors crea un *connector* invocando a sua volta la funzione *create_connector* di cui si riporta il codice (di per sé semplice e poco significativo) giusto per mostrare l'uso dell'endpoint REST suddetto:

```
1 def create_connector(host: str, payload: dict):
2     URL: str = host + "connectors/"
3     response = requests.post(url=URL, json=payload)
4     if response.status_code != status.HTTP_201_CREATED:
5         raise Exception(response.json())
```

A conclusione di questo paragrafo, si fa notare che una configurazione di test non viene rimossa in automatico una volta che l'ambiente di test in esecuzione viene fermato; ciò va fatto, invece, manualmente avviando un semplice script che cancella tutti i *connectors*, i *kafka topics* e le collezioni aventi un certo prefisso. In questo modo è possibile avere a disposizione i dati che sono stati prodotti nell'ambiente di test anche al termine del test medesimo, al fine di poter effettuare ulteriori analisi per capire se i risultati prodotti sono quelli che ci si aspettava.

4.4 Semplificazione dell'esecuzione dei test

La possibilità di poter eseguire dei test in un ambiente apposito ha reso più facile la rilevazione di alcuni potenziali problemi. Per semplificare ulteriormente l'esecuzione di un singolo test, sono state fornite due caratteristiche aggiuntive:

1. la possibilità di dichiarare quali *agents* eseguire in un dato test;
2. la presenza di un'applicazione" *playground*" per *faust*.

4.4.1 La selezione degli agents

Testare più *stream processors* in esecuzione non è banale in quanto essi spesso (almeno su iChain) utilizzano in input anche dei dati prodotti da un altro *stream processor*: in caso di un qualche problema/errore nell'output ultimo, non sempre è possibile identificare con certezza quale degli *agents* coinvolti ha dato origine al problema.

L'ideale sarebbe quindi poter indicare espressamente quale o quali *agents* avviare in un dato test, al fine di potersi concentrare nello specifico sul singolo o sui singoli *agents* e analizzare, per verificare che tutto funzioni, come essi svolgano il proprio specifico compito, senza doversi preoccupare di ciò che altri *agents* potrebbero fare. Quando si crea un'applicazione *faust* è possibile specificare il parametro *autodiscover*, il quale può assumere tre valori possibili:

1. il valore booleano *true*, che indica di cercare gli *agents* da avviare in tutti i *packages* dell'applicazione quando essa è avviata;
2. il valore booleano *false*, che indica di non cercare alcun *agent*. È il valore di default;
3. una lista di *packages* in cui cercare gli *agents* da avviare all'avvio dell'applicazione.

Precedentemente al presente lavoro, l'applicazione *faust* di iChain era avviata con *autodiscover=true*, con il conseguente avvio di tutti gli *agents* che, in fase di produzione, è chiaramente ciò che si vuole avvenga. In un test invece, come specificato sopra, è spesso preferibile poter avviare gli *agents* singolarmente o poter definire quali avviare. Per far ciò, la scelta apparentemente più ovvia è quella di indicare nel parametro *autodiscover* i *packages* in cui cercare gli *agents* da avviare. Di per sé, ciò produce l'effetto di non avviare tutti gli *agents*, ma soltanto quelli che si trovano nei *packages* elencati, il che causerebbe il non avvio di tutti gli *agents* anche in un ambiente di produzione.
Pertanto, per soddisfare entrambe le necessità, si è scelto di percorrere i passi seguenti:

- si è settato il parametro *autodiscover* con il valore booleano *false*. In questo modo, come si è detto, all'esecuzione dell'applicazione *faust* non sarà avviato alcun *agent* in automatico;
- si è creato un file di configurazione, esterno al codice, in cui sono indicati i *packages* in cui cercare gli *agents* da avviare;
- in fase di inizializzazione dell'applicazione *faust*, quindi prima del suo avvio, si procede come segue:

1. si legge dal file di configurazione la lista di *packages* da avviare;
2. tale lista viene passata come parametro alla funzione *discover*, la quale è fornita direttamente dalla libreria *faust*. L'esecuzione di tale funzione farà in modo che l'applicazione *faust*, una volta avviata, cercherà gli *agents* da eseguire nei *packages* letti in precedenza dal file di configurazione.

Grazie alla presenza di un file di configurazione esterno al codice è dunque possibile avviare, in un ambiente di test, solo gli *agents* desiderati semplicemente indicando la lista dei *packages* che li contengono; allo stesso tempo, in un ambiente di produzione, è possibile avviare tutti gli *agents* semplicemente inserendo tutti i *packages* nel file di configurazione (in realtà si è preferito fare in modo che, se il file di configurazione è vuoto, ciò venga interpretato come una richiesta di cercare gli *agents* in tutti i *packages*.

4.4.2 Un "playground" per Faust

Il termine *playground* si traduce letteralmente come "parco giochi". Nel campo dell'informatica, un'applicazione "parco gioco" può essere vista come un'applicazione su cui sperimentare ed esplorare le funzionalità di un certo linguaggio di programmazione o di una libreria. Nel contesto di questa tesi, è stato implementato un piccolo playground in cui provare le funzionalità della libreria *faust*.
La libreria *faust* è, infatti, ricca di funzionalità ed è anche un elemento importante di iChain in quanto usata per implementare gli *agents* che poi sono usati dai plugins di iChain sia interni sia privati (i.e. realizzati specificatamente per un'azienda, come si è visto).
La presenza di un'applicazione *playground* per *faust* è quindi utile per poter fare delle prove e studiare al meglio le funzionalità della libreria in "un'area sicura", ovvero senza dover toccare il codice dell'applicazione (potenzialmente sempre più complesso nel tempo) in cui sono eseguiti gli *agents* veri.
L'applicazione *playground* implementata è costituita da un singolo *agent* che legge da un *kafka topic* un insieme di coppie chiave-numero (numero generato randomicamente) e scrive su una *table* (su *faust*, una *table* è un dizionario in memoria RAM che è supportato da un personale *kafka topic* al fine di ottenere persistenza e tolle-

ranza agli errori [40]) la somma dei numeri ricevuti per ogni chiave. Per esempio, se vengono ricevute le coppie ("key1", 11), ("key2, 3"), ("key1", 4), ("key1", 3), ("key3", 96), ("key2", 5), la *faust table* conterrà le seguenti coppie: ("key1", 18), ("key2", 8), ("key3", 96).
Quest'applicazione *playground* ha permesso di testare meglio la serializzazione sul *kafka topic* associato a una *faust table* che non funzionava adeguatamente nel modo in cui veniva fatta nell'applicazione "reale".

4.5
Problemi rilevati grazie alle esecuzioni in ambienti di test

Oltre a una serie di bug "minori", grazie all'esecuzione degli *agents* in ambienti di test sono stati rilevati alcuni problemi degni di nota. Essi sono:

1. l'*agent* incaricato di trasmettere gli eventi EPCIS ai partner (in base alle *regole di inoltro* definite) inoltrava l'evento senza inserire il campo *data_ origin*, con la conseguente impossibilità di riconoscere gli eventi EPCIS prodotti da quelli ricevuti da terzi.
   Il problema è stato risolto, in primo luogo, facendo in modo che l'*agent* aggiunga il campo *data_ origin* all'evento EPCIS prima di inoltrarlo. Per correggere gli eventi EPCIS già memorizzati sui database è stato anche necessario, in secondo luogo, scrivere uno script di fix che fosse in grado di determinare chi avesse generato un certo evento EPCIS per poter, così, aggiungere il campo *data_ origin*. Il criterio adottato per determinare chi ha generato un evento EPCIS è stato il seguente:

   (a) si sono raggruppati gli eventi EPCIS per campo *event_ id*;

   (b) per ogni raggruppamento:

   - se il gruppo contiene solo un elemento, allora l'evento EPCIS non è stato inoltrato ad alcun partner e quindi il campo data_origin è uguale al campo object_owner (i.e. il dominio dell'azienda che possiede l'evento);

- altrimenti, si vede quali aziende hanno un template di eventi con lo stesso nome del template d'evento da cui è stato generato l'evento EPCIS: se vi è una sola azienda, allora al campo data_origin sarà assegnato il dominio di quell'azienda;
- altrimenti si guarda quali aziende hanno generato eventi EPCIS nel luogo in cui è stato generato lo specifico evento EPCIS a cui si sta cercando di assegnare una data_origin. Se vi è solo un'azienda, al campo data_origin viene assegnato il dominio di quell'azienda.
- altrimenti non è possibile assegnare una data_origin. Ovviamente il criterio di assegnazione è quello descritto in quanto, nei database esistenti, quest'ultimo caso non si è verificato (i.e. è stato possibile assegnare a tutti gli eventi EPCIS esistenti un campo data_origin);

2. la cattiva serializzazione sul *kafka topic* associato a una *faust table* che, come si è visto, è stata risolta grazie all'ausilio dell'applicazione *playground* per *faust*;

3. un *agent* restava bloccato all'infinito se provava a consumare un messaggio da un *kafka topic* che ancora non era stato completamente creato. Il problema è stato risolto programmando ogni *agent* affinché attenda che i *kafka topics* da cui legge siano stati completamente creati, prima di provare a consumare dei messaggi da essi.

5

Meccanismi di estensione dello standard EPCIS

Lo standard EPCIS è molto flessibile, poiché ha l'obiettivo di riuscire ad adattarsi a svariate situazioni e poter essere usato da molteplici aziende. Ogni azienda, però, può avere le proprie specificità nei propri flussi produttivi che spesso hanno poco o nulla in comune con le altre aziende. Lo standard EPCIS, dunque, fornisce anche alcuni *meccanismi di estensione* per consentire alle aziende che usano tale standard di poter rappresentare situazioni relative a un proprio specifico processo di business. Nel corso di questo capitolo si discuteranno:

- i tipi di estensione EPCIS;
- *JSON-LD* e *SHACL*;
- come le estensioni EPCIS sono gestite dalla piattaforma iChain.

5.1
I tipi di estensione EPCIS

EPCIS è uno standard *estensibile* in quanto, oltre a fornire una serie di elementi standard e generali, fornisce anche dei meccanismi con cui l'insieme di elementi base può essere esteso per soddisfare scopi specifici di una data azienda o di un'area applicativa. Questa estensibilità dello standard EPCIS permette anche allo standard stesso, inoltre, di evolversi e crescere nel tempo in un modo del tutto naturale.
Peraltro, lo standard EPCIS è anche modulare in quanto è possibile specificare diverse parti dello standard, incluse le estensioni, in diversi documenti.
Esistono vari tipi di estensione:

- aggiunta di un nuovo tipo di evento EPCIS;
- aggiunta di un nuovo campo a un tipo di evento EPCIS già esistente;
- aggiunta di un nuovo tipo di vocabolario;

- aggiunta di un nuovo attributo dei *master data*;
- aggiunta di un nuovo attributo *ilmd*;
- aggiunta di un nuovo vocabolo in un vocabolario già esistente.

La tabella 5.1 riporta i contesti in cui può essere emessa un'estensione, l'organizzazione coinvolta e, per ciascuno tipo di estensione sopra menzionato, se è in genere presente oppure no.

Come viene diffusa l'estensione	**Organizzazione responsabile**	**Nuovo tipo di evento**	**Nuovo campo per un evento**	**Nuovo tipo di vocabolario**	**Nuovo attributo master data o ilmd**	**Nuovo vocabolo**
Nuova versione dello standard EPCIS	GS1 EPCIS/CBV MSWG	Sì	Sì	Sì	Occasionalmente	Raramente
Nuova versione dello standard CBV	GS1 EPCIS/CBV MSWG	No	No	No	Sì	Sì
Standard di applicazione GS1 per un settore specifico	GS1 Application Standard Working Group per uno specifico settore	Raramente	Raramente	Occasionalmente	Sì	Sì
Documento di raccomandazioni di un settore specifico all'interno di un'area geografica	Organizzazione membro di GS1	Raramente	Raramente	Occasionalmente	Sì	Sì
Specifica di interoperabilità di un gruppo privato	Consorzio industriale o un gruppo di utenti finali esterni a GS1	Raramente	Raramente	Occasionalmente	Sì	Sì
Aggiornamento dei *master data* tramite *EPCIS Query*	Utente finale individuale	Raramente	Raramente	Raramente	Raramente	Sì

Tabella 5.1: frequenza di rilascio dei vari tipi di estensione dello standard EPCIS in alcuni contesti [12].

Siccome a partire dalla versione 2.0 EPCIS utilizza JSON come formato principale per rappresentare i dati, usa *JSON-LD* per definire l'estensione.

5.2 JSON-LD

JSON-LD (*JSON for Linked Data*) è un formato di interscambio di *Linked Data* che fa uso di JSON [41].

I *Linked Data* sono *dati strutturati* (i.e. meta-informazioni) che possono essere interconnessi con altri dati strutturati. Nei *Linked Data*, gli *URI* (*Uniform Resource Identifier*) sono usati come identificativo univoco e non ambiguo dei dati.

Una rappresentazione *Linked Data* può essere rappresentata in vari formati, come per esempio *RDF* (*Resource Description Framework*), *RDFS* (*RDF Schema*), *OWL* (*Web Ontology Lanaguage*), *SKOS* (*Simple Knowledge Organization System*) e, in particolare, *JSON-LD*.

Lo scopo dei *Linked Data* è far sì che i sistemi informatici siano capaci di leggere e interpretare direttamente le informazioni disponibili sul web, estraendo i dati da varie fonti seguendone i collegamenti.

La figura 5.1 illustra graficamente i *Linked Data*.

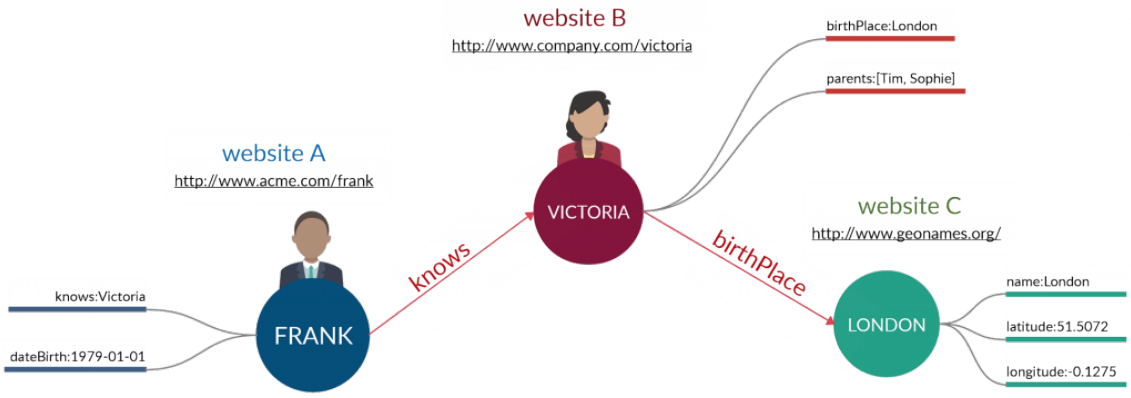

Figura 5.1: una schematizzazione dei *Linked Data* [42]

5.2.1 Forme di JSON-LD

Come spesso avviene per molti formati di dati, anche in *JSON-LD* i dati possono essere descritti in più forme diverse; in particolare è possibile avere quattro rappresentazioni diverse [41]:

1. *expanded form* (forma espansa). L'*espansione* è quel processo per il quale a un documento *JSON-LD* viene applicato il contenuto del campo *@context* in modo tale che tutti gli *IRI* (Internationalized Resource Identifier), i tipi e i valori siano espansi potendo così, di fatto, rimuovere il campo *@context* dal documento;

2. *compacted form* (forma compatta). La *compattazione* consiste nell'applicare un contesto a un documento *JSON-LD* per abbreviare gli *IRI* in termini o per compattare gli *IRI* e i valori *JSON-LD*, espressi in forma espansa, in valori più semplici (e.g. stringhe o numeri). I vantaggi di avere un documento *JSON-LD* in forma compatta sono:

   (a) maggiore semplicità nel lavorare con il documento, in quanto esso esprime i dati con termini specifici per una data applicazione;

   (b) maggiore comprensione del documento quando questo è letto da una persona;

3. *flattened form* (forma appiattita). In un documento "appiattito", tutte le proprietà di un singolo nodo sono raggruppate in un dizionario e tutti i nodi vuoti sono etichettati con un identificatore di nodo vuoto;

4. *framed form* (forma incorniciata). Il *framing* consiste nell'usare un documento per comporre un altro documento *JSON-LD*.

Di seguito, si riporta un esempio di uno stesso documento *JSON-LD* rappresentato nelle quattro forme sopra elencate:

- *Expanded JSON-LD*:

```
1 [
2   {
```

```
    "@type": [
      "http://schema.org/Riccardo"
    ],
    "http://schema.org/jobTitle": [
      {
        "@value": "Computer engineer"
      }
    ],
    "http://schema.org/name": [
      {
        "@value": "Tedesco"
      }
    ],
    "http://schema.org/url": [
      {
        "@id": "http://www.riccardotedesco.com"
      }
    ]
  }
]
```

- *Compacted JSON-LD*:

```
{
  "@context": "http://schema.org/",
  "type": "Riccardo",
  "jobTitle": "Computer engineer",
  "name": "Tedesco",
  "url": "http://www.riccardotedesco.com"
}
```

- *Flattened JSON-LD*:

```
{
  "@context": "http://schema.org/",
  "@graph": [
    {
      "id": "_:b0",
      "type": "Riccardo",
      "jobTitle": "Computer engineer",
      "name": "Tedesco",
      "url": "http://www.riccardotedesco.com"
    }
  ]
}
```

- *Framed JSON-LD*:

```
{
  "@graph": [
    {
      "@type": "http://schema.org/Riccardo",
      "http://schema.org/jobTitle": "Computer
         engineer",
      "http://schema.org/name": "Tedesco",
      "http://schema.org/url": {
        "@id": "http://www.riccardotedesco.com"
      }
    },
    {
      "@id": "http://www.riccardotedesco.com"
    }
  ]
}
```

5.2.2 JSON-LD e EPCIS 2.0

Con il supporto al formato JSON, EPCIS 2.0 non poteva non scegliere la rappresentazione *JSON-LD* per rappresentare i *Linked Data*.
EPCIS 2.0 fa uso delle seguenti parole chiave di *JSON-LD* [12]:

- *@id*, che può essere usato o per identificare un oggetto o come riferimento a un identificativo di oggetto;
- *@type*, è usato per specificare in quale tipo deve essere convertito un valore (che è rappresentato come di tipo stringa);
- *@vocab*, che è usato per specificare il vocabolario da usare di default per interpretare i termini;
- *@context*, che è un contenitore di oggetti di dati in cui possono essere specificati, sia localmente al contesto stesso sia referenziati tramite URL, i seguenti elementi:
  - un vocabolario di default, usando la parola chiave *@vocab*;
  - espansioni per i prefissi *CURIE* (*Compact URI Expression*). Una *CURIE* serve per definire un'etichetta per una URL, nella forma *label:URL*. Un possibile esempio è: *"xsd": "http://www.w3.org/2001/XMLSchema#"*, in cui xsd sarà un sinonimo del URL indicato. Ogni volta che si incontrerà, quindi, qualcosa del tipo *xsd:dateTimeStamp* questo dovrebbe essere espanso in *http://www.w3.org/2001/XMLSchema#dateTimeStamp* [43];
  - le mappature dei campi a tipi di dati diversi dalle stringhe. Possibili esempi sono:
    - "eventTime": "@type":"xsd:dateTimeStamp";
    - "quantity": "@type":"xsd:decimal";
    - "epcList": "@type":"@id";
  - degli alias per le parole chiave di *JSON-LD*. Possibili esempi sono:
    - "isA": "@type";
    - "id":"@id".

L'uso di *JSON-LD* in EPCIS 2.0 consente di definire delle estensioni per un evento EPCIS. Ogni estensione, infatti, può semplicemente essere definita in un contesto *JSON-LD* e rappresentata tramite *CURIE*.

Un esempio di *JSON-LD* contenente un'estensione potrebbe essere:

```
{
  "@context": ["https://gs1.github.io/EPCIS/epcis-context
     .jsonld",{"example": "http://ns.example.com/epcis/"}
     ],

  "id": "_:document1",
  "isA": "EPCISDocument",
  "schemaVersion": "2.0",
  "creationDate":"2005-07-11T11:30:47.0Z",
  "epcisBody": {
   "eventList": [
     {
      "eventID": "ni:///sha-256;87b5f18a69993f0052046d468
         7dfacdf48f7c988cfabda2819688c86b4066a49?ver=CBV2
         .0",
      "isA": "AggregationEvent",
      "eventTime": "2022-02-19T14:58:56.591Z",
      "eventTimeZoneOffset": "+02:00",
      "parentID":"urn:epc:id:sscc:0614141.1234567890",
      "childEPCs":["urn:epc:id:sgtin:0614141.107346.2017"
         ,"urn:epc:id:sgtin:0614141.107346.2018"],
      "action": "OBSERVE",
      "bizStep": "receiving",
      "disposition": "in_progress",
      "readPoint": {"id": "urn:epc:id:sgln:0614141.00777.
         0"},
      "bizLocation": {"id": "urn:epc:id:sgln:0614141.0088
```

```
          8.0"},
22        "childQuantityList": [
23          {"epcClass":"urn:epc:idpat:sgtin:4012345.098765
              .*","quantity":10},
24          {"epcClass":"urn:epc:class:lgtin:4012345.012345.9
              98877","quantity":200.5,"uom":"KGM"}
25        ],
26        "example:myField":"Example of a vendor/user
            extension"
27      }
28    ]
29  }
30 }
```

In questo esempio è possibile vedere che vi è un contesto *JSON-LD* in cui si fa riferimento al namespace EPCIS tramite una URL e in cui si definisce la CURIE *"example": "http://ns.example.com/epcis/"*. In questo caso, *example* potrebbe tranquillamente essere l'etichetta rappresentante un'estensione il cui schema è reperibile all'indirizzo *http://ns.example.com/epcis/*.
Alla riga 26 è quindi possibile vedere un uso di tale estensione, che introduce all'evento EPCIS un campo non definito dallo standard stesso.
Chiunque riceverà questo evento EPCIS, potrà riconoscere l'estensione semplicemente osservando il contesto *JSON-LD* e potrà successivamente decidere, quindi, se tenerla in considerazione - qualora di suo interesse - oppure scartarla.

5.3 SHACL

SHACL (*Shapes Constraint Language*) è un linguaggio per la validazione dei grafi *RDF* a partire da un insieme di condizioni prestabilite, a loro volta espresse sotto forma di grafi *RDF* [44].
SHACL richiede in input due parametri:

1. uno *shapes graph*, di solito o in formato *turtle* o in formato *JSON-LD*, che è un documento contenente:

    - un insieme di condizioni volte a definire il tipo (stringhe, interi) che quella proprietà deve avere e limitare l'insieme di valori possibili che una proprietà può avere;
    - la definizione dei modelli di corrispondenza delle stringhe;
    - combinazioni logiche di determinati valori.

2. un *data graph*, cioè un documento contenente dei dati strutturati *RDF* (e.g. un documento *JSON-LD*).

A partire da uno *shapes graph* e da un *data graph*, *SHACL* verifica se il *data graph* soddisfa quanto lo *shapes graph* esprime e fornisce in output, di conseguenza, un report sul risultato della validazione; nel caso in cui non ci sia corrispondenza tra il *data graph* e lo *shapes graph*, il report conterrà anche l'elenco dei problemi riscontrati e che hanno fatto fallire la validazione.
SHACL può quindi essere usato, per esempio, per validare un documento *JSON-LD*, verificando che esso soddisfi un certo schema (il documento *JSON-LD* da validare è il *data graph*, lo schema è lo *shapes graph*). Questo, per esempio, potrebbe essere usato per verificare che un'estensione EPCIS contenuta in un evento EPCIS soddisfi un certo schema in cui tale estensione è definita.

5.4
Implementazione dei meccanismi di estensione su iChain

In questo lavoro di tesi, si è deciso di abbozzare su iChain una prima forma di implementazione dei meccanismi di estensione dello standard EPCIS in modo tale che la piattaforma sia in grado di gestire anche eventuali campi o dati che non sono codificati dallo standard EPCIS ma che un'azienda può, comunque, ritenere necessari per il proprio contesto di business.

5.4.1 Analisi della forma di JSON-LD da usare su iChain

Il fatto che la piattaforma iChain faccia uso di *MongoDB* come database si concilia alla perfezione con il bisogno di dover usare *JSON-LD* per le estensioni. Infatti, poiché in *MongoDB* i documenti sono memorizzati in un formato *JSON-like*, non vi è alcun problema a memorizzare un *JSON-LD* in un documento su *MongoDB*.
Si è scelto di memorizzare su *MongoDB* i *JSON-LD* in forma *compatta*, in quanto questa forma garantisce un minor spazio richiesto per memorizzare un *JSON-LD* e fornisce una migliore leggibilità. In questo caso, quindi, il campo *@context* di un *JSON-LD* rappresentante un'estensione sarà del tipo *etichetta:URL* e ogni attributo dell'estensione avrà una chiave del tipo *etichetta:attributo*.
Prima di scegliere di usare la forma compatta, ci si è chiesti se non fosse stato meglio utilizzare, invece, la forma espansa per il *JSON-LD* da memorizzare su *MongoDB* in quanto, nel caso si fosse dovuto creare un indice su un attributo di un'estensione di un evento EPCIS, la forma espansa sarebbe stata sicuramente migliore dato che l'attributo sarebbe già rappresentato nella forma *URL:attributo*. Tuttavia, alla fine si è giunti alla conclusione che creare un indice su un attributo di un'estensione di un evento EPCIS avrebbe poco senso poiché si tratterebbe di un indice su pochi eventi EPCIS (un'estensione, in realtà, riguarda solo pochi eventi EPCIS in quanto è necessaria solo in uno specifico contesto). Qualora dovesse poi risultare necessario creare un indice su un qualche attributo di un'estensione di un evento EPCIS, si potrà ovviare semplicemente usando una *tabella di indirezione*. In definitiva, si è optato per l'uso della forma compatta.

5.4.2 Ampliamento del data model di un evento EPCIS

Il data model di un evento EPCIS contiene già un campo *extensions* nel quale sono rappresentati i seguenti attributi:

- *quantityList* che, per ogni tipo di oggetto coinvolto in un evento EPCIS, ne indica la quantità;

- *childQuantityList* che, per ogni oggetto figlio coinvolto in un evento EPCIS di tipo *AggregationEvent* o *AssociationEvent*, ne specifica la quantità;

- *sourceList*, che contiene la sorgente di un evento EPCIS di tipo *TrasactionEvent*;
- *destinationList*, che contiene la destinazione di un evento EPCIS di tipo *TrasactionEvent*;
- *ilmd*, che contiene *i master data*.

Tali attributi sono tutti elementi opzionali di un evento EPCIS (in alcuni casi la loro presenza dipende anche dal tipo di evento EPCIS). Queste "estensioni", quindi, sono di fatto delle estensioni previste e definite dallo standard e, per tale motivo, si è ritenuto non opportuno inserire quelle relative a un dato contesto di business direttamente nel campo *extensions* del data model di un evento EPCIS.
Dunque, al data model di un evento EPCIS è stato aggiunto il campo *proprietary_extensions* con il compito di rappresentare, per ogni evento EPCIS, una lista opzionale di estensioni relative a specifici contesti di business e, perciò, non codificate direttamente dallo standard EPCIS. Ogni elemento di tale lista, contiene i seguenti campi:

- il campo *context* (di fatto, il campo *@context* di un *JSON-LD*) che a sua volta contiene:
  - il campo *label*, che è una stringa rappresentante l'etichetta che identifica una particolare estensione;
  - il campo *url*, che contiene l'indirizzo in cui è possibile reperire lo schema *RDF* associato all'estensione.
- il campo *data_graph*, che è un dizionario contenente una serie di elementi *<label:attributo, valore>* che rappresentano i vari attributi dell'estensione.

Ogni elemento della lista *proprietary_extensions* contiene, quindi, una specifica estensione di un evento EPCIS che rappresenta un certo contesto di business.

5.4.3 La gestione delle estensioni di un evento EPCIS

Per gestire un'estensione di un evento EPCIS, al momento ci si limita a far verificare allo *stream processor* che cattura i nuovi eventi EPCIS se questi contengono

almeno un elemento nella lista *proprietary_ extensions*. In caso affermativo, per ogni estensione si converte il campo *url* nella rappresentazione *url-safe* e si usa tale rappresentazione come nome del *kafka-topic* su cui pubblicare l'evento EPCIS che possiede un'estensione.

In questo modo, eventuali plugins addizionali implementati per una specifica azienda potranno poi consumare i messaggi dai *kafka topics* contenenti gli eventi EPCIS aventi estensioni di interesse per l'azienda medesima al fine di poter poi analizzare tali eventi EPCIS nella maniera ritenuta più opportuna a soddisfare le esigenze dell'azienda per la quale è stato implementato il plugin.

Naturalmente, ogni plugin dovrebbe come prima cosa verificare la validità delle estensioni mediante l'utilizzo di *SHACL*.

Di seguito si riporta il frammento di codice, che è parte dell'*agent* che cattura i nuovi eventi EPCIS, che realizza le operazioni descritte sopra.

```
1 if event.data_origin == event.object_owner:
2     for proprietary_extension in event.proprietary_extensions:
3         b64_url = urllib.parse.quote(proprietary_extension.context.
              url, safe='')
4         private_topic = app.topic(settings.PREFIX_STR + b64_url)
5         await private_topic.send(key=event_id, value=json_util.
              dumps(event))
```

6

Conclusioni

In seguito a questo percorso di tesi, la piattaforma iChain è stata migliorata ed è quindi, allo stato attuale, in grado di:

- gestire il nuovo tipo di evento AssociationEvent che, come specificato precedentemente, è stato introdotto nella versione 2.0 dello standard EPCIS;
- gestire attributi specifici per un dato dominio di business e non rappresentati direttamente dallo standard EPCIS.

Inoltre, l'ambiente di test che è stato messo a punto in questa occasione, oltre ad aver consentito di verificare la corretta gestione dei due aspetti sopra riportati, potrà anche consentire in futuro di aggiungere nuovi eventi che dovessero essere previsti in versioni successive dello standard EPCIS. In definitiva, per concludere, si può ragionevolmente ritenere che tutti gli obiettivi inizialmente prefissati siano stati raggiunti. Un possibile sviluppo futuro, oltre all'inserimento di nuove tipologie di eventi, potrà essere quello di aggiungere alla piattaforma un editor di estensioni di eventi EPCIS per consentire, a ogni utente registrato sulla piattaforma, di creare autonomamente le proprie estensioni (al momento ciò è consentito solo all'amministratore della piattaforma stessa).

A

I template di eventi EPCIS

Un evento EPCIS è costituito da un ampio insieme di attributi più o meno articolati. Dal punto di vista di un utente, fornire tutti i parametri necessari può essere fastidioso e al tempo stesso non scontato, in quanto è richiesta un'approfondita conoscenza degli standard EPCIS e CBV.
La piattaforma iChain, ampiamente consapevole del fatto che pure il miglior servizio, se non è *user-friendly*, non potrà mai avere successo e abbracciare una vasta utenza, viene ancora una volta incontro alle esigenze dei fruitori del servizio fornendo loro la possibilità di usare i cosiddetti "*modelli di eventi EPCIS*" (*EPCIS event template*) [2]. I template di eventi EPCIS richiesti da un'azienda sono in genere realizzati in seguito a incontri preliminari tra alcuni rappresentanti dell'azienda stessa e l'esperto dello standard EPCIS di iChain. Durante tali incontri, l'esperto dello standard EPCIS si farà specificare dal cliente quali siano per lui i processi produttivi di interesse definendo, quindi, la struttura (ovvero definendo quali sono gli attributi EPCIS per quel dato processo produttivo che sono noti a priori e quali invece vanno inseriti necessariamente) di ogni template di evento necessario a rappresentare quei processi produttivi; dopodiché gli esperti informatici di iChain provvederanno a inserire nella piattaforma i template di eventi definiti per quello specifico cliente, il quale potrà quindi generare gli eventi EPCIS di interesse semplicemente usando un appropriato template di evento definito.
Un template di evento EPCIS, quindi, richiede che l'utente fornisca in input solamente un insieme di campi necessari e non predicibili a priori, dopodiché l'evento EPCIS sarà automaticamente costruito dalla piattaforma combinando le informazioni fornite in input dall'utente con alcune informazioni precedentemente immagazzinate nel corpo del template.
Inoltre, ogni template di evento ha un nome che deve essere univoco all'interno di un dato dominio aziendale registrato sulla piattaforma iChain [2]. Questo nome del template, che poi sarà propagato anche agli eventi EPCIS generati a partire dal

template stesso, consente di poter identificare meglio la situazione di un particolare processo produttivo che un evento EPCIS rappresenta. Gli standard EPCIS e CBV, infatti, definiscono una semantica generica e questa genericità non permette di identificare a prima vista cosa un evento EPCIS rappresenta senza esaminarne in profondità la struttura.
Un template di evento EPCIS, quindi, contiene le seguenti informazioni:

- un nome atto a identificare, come si è detto poc'anzi, cosa un particolare evento EPCIS rappresenta;
- il tipo di evento EPCIS, che può assumere uno dei valori visti nel paragrafo 2.8.1, ovvero *ObjectEvent*, *AggregationEvent*, *TransformationEvent*, *TransactionEvent* e il recentissimo (al momento della stesura di questa tesi) *AssociationEvent*;
- un campo per ognuna delle quattro dimensioni, ovvero le dimensioni *4W* *what*, *when*, *where*, *why* (c.f.r. par. 2.8.2);
- un campo per i cosiddetti *ilmd* (*Instance/Lot master data*), ovvero per i *master data* (c.f.r. par. 2.8.3).

Ognuna delle dimensioni delle *4W* può contenere una lista di oggetti JSON, ognuno dei quali è composto dai seguenti campi [2]:

- il campo *name*, che è obbligatorio ed è una stringa rappresentante il nome dell'attributo EPCIS da valorizzare (e.g. *epcList*, *childEPCs*, *parentID*, *bizLocation*);
- il campo *type*, che è obbligatorio e rappresenta il tipo (e.g. *EPCList*, *QuantityList*, *date*) dell'attributo EPCIS da valorizzare (N.B.: non va confuso con il campo *type* di un evento EPCIS) e serve al codice che, a partire da questo template di evento, dovrà generare il corrispondente evento EPCIS;
- il campo *value*, che indica il valore assunto dall'attributo EPCIS valorizzato;
- il campo *input_type*, che è obbligatorio e indica il valore che sarà scritto nel campo *value*. Può assumere 5 valori:

1. *external*, che indica che il valore dell'attributo EPCIS dovrà essere ricevuto esternamente, per esempio dovrà essere fornito in input dall'utente. In questo caso il campo *value* è settato a *null*;
2. *fixed*, che indica che il valore dell'attributo è stato definito al momento della creazione del template di evento e non potrà variare nel tempo;
3. *computed*, che indica che il valore dell'attributo, se non è fornito direttamente dall'utente, sarà calcolato automaticamente dalla piattaforma (e.g. la data in cui si è verificato l'evento EPCIS alla quale, a meno di indicazioni particolari, verrà assegnata la data corrente);
4. *partially-fixed*, che indica che il valore dell'attributo è in parte stato definito al momento della creazione del template di evento e in parte dovrà essere fornito esternamente. Tale caso è possibile solo per gli attributi *bizTransactionList*, *sourceList* e *destinationList*;
5. *client-computed*, che indica che il valore dell'attributo deve essere fornito dal client ma in maniera automatizzata (cioè, non può essere digitato a mano in un form);

- il campo *alias*, che è opzionale e definisce un sinonimo, potenzialmente più comprensibile all'utente, del campo *name*;
- il campo *function_ name*. Tale campo è presente solo se il campo *input_ type* ha il valore *computed* e rappresenta la funzione che il back-end della piattaforma iChain dovrà usare per determinare il valore dell'attributo EPCIS;
- il campo *precedence_ value*, che è un campo opzionale che, se presente, indica che tale evento è valido se e solo se nella piattaforma è stato generato in precedenza un altro evento EPCIS avente i valori per le dimensioni *what* e *why* specificati in questo campo;
- il campo *whitelist*, che è opzionale e, se presente, definisce una lista di valori ammissibili per le dimensioni *what* e *where*;
- il campo *strict*, che è opzionale e ha ragione di esistere solo se è definito il campo *whitelist*. Tale campo, se presente, indica che le dimensioni *what* e

where non possono contenere alcun valore diverso da quelli presenti nella lista di valori indicata nel campo *whitelist*;

- il campo *optional*, che è un campo booleano di default inizializzato con il valore booleano *false*. Se vale *true* e il campo *input_ type* ha il valore *external*, allora l'utente dovrà obbligatoriamente fornire il dato.

Anche il campo *ilmd* può contenere una lista di oggetti JSON, ognuno dei quali è composto dai seguenti campi [2]:

- il campo *ns*, che è obbligatorio e indica il namespace a cui appartiene il *master data*;
- il campo *name*, che è obbligatorio e indica il nome del *master data*;
- il campo *input_ type*. Vale quanto detto in precedenza, anche se nel caso dei *master data* tale campo solitamente è uguale a *external*;
- il campo *value*, che indica il valore del *master data*;
- il campo *default*, che è opzionale e, se presente, indica il valore predefinito da assegnare al *master data* nel caso in cui questo non venga fornito;
- il campo *optional*, che è di tipo di booleano e di default vale *true*. Indica se un determinato master data deve essere sempre fornito per gli eventi facenti riferimento a questo template di evento (i.e. *optional* = *false*) oppure se l'evento risulta valido anche senza di esso (i.e. *optional* = *true*).

I template di eventi EPCIS sono, quindi, una caratteristica fornita da iChain che consente a chiunque di poter utilizzare la piattaforma e generare eventi EPCIS senza dover conoscere in modo approfondito i dettagli dello standard.

Bibliografia

[1] Mecalux. Supply chain: cos'è e come funziona la catena di approvvigionamento. https://www.mecalux.it/blog/supply-chain-cos-e. Accessed: 23/02/2022.

[2] Ennio Riccobene. Sviluppo backend di una piattaforma per la trasparenza e la tracciabilità nelle filiere produttive. *Tesi di laurea magistrale, Ingegneria Informatica, Politecnico di Torino*, a.a. 2019/20.

[3] Wiseside s.r.l. ichain: helicopter view. https://wiseside.com/theme/canvas-6.5.3/images/new/ichain_1.jpg, 2021. Accessed: 31/01/2022.

[4] GS1. Gs1 - calcolo della cifra di controllo. https://gs1it.org/assistenza/calcolo-cifra-di-controllo/, . Accessed: 31/01/2022.

[5] GS1. Gln - global location number. https://gs1it.org/assistenza/standard-specifiche/gln/, . Accessed: 31/01/2021.

[6] GS1. Gtin - global trade item number. https://gs1it.org/assistenza/standard-specifiche/gtin/, . Accessed: 31/01/2022.

[7] GS1. Gdti - global document type identifier. https://gs1it.org/assistenza/standard-specifiche/gdti-global-document-type-identifier/, . Accessed: 31/01/2022.

[8] GS1. Sscc - serial shipping container code. https://gs1it.org/assistenza/standard-specifiche/sscc/, . Accessed: 31/01/2022.

[9] GS1. Grai - global returnable asset identifier. https://gs1it.org/assistenza/standard-specifiche/grai/, . Accessed: 31/01/2022.

[10] GS1. Giai - global individual asset identifier). https://www.gs1.org/standards/id-keys/global-individual-asset-identifier-giai, . Accessed: 31/01/2022.

[11] GS1. Epcis and cbv implementation guideline. using epcis and cbv standards to gain visibility of business processes. release 1.2. https://www.gs1.org/sites/default/files/docs/epc/EPCIS-Standard-1.2-r-2016-09-29.pdf, September 2016. Accessed: 01/02/2022.

[12] GS1. Standards development public reviews. https://www.gs1.org/standards/development-work-groups/public-reviews#EPCISCBV_Public-Review-2021, . Accessed: 19/02/2022.

[13] GS1. Epcis and cbv implementation guideline. using epcis and cbv standards to gain visibility of business processes. https://www.gs1.org/standards/epcis-and-cbv-implementation-guideline/current-standard#1-Introduccin+undefined, Febrary 2017. Accessed: 01/02/2022.

[14] GS1. Core business vocabulary standard (cbv standard), release 1.2.2. https://www.gs1.org/sites/default/files/docs/epc/CBV-Standard-1-2-2-r-2017-10-12.pdf, October 2017. Accessed: 01/02/2022.

[15] Samuel Colvin. Pydantic. https://pydantic-docs.helpmanual.io/. Accessed: 06/02/2022.

[16] Tom Collings. Controller-service-repository. https://tom-collings.medium.com/controller-service-repository-16e29a4684e5, August 2021. Accessed: 08/02/2022.

[17] Tom Collings. Controller-service-repository. https://miro.medium.com/max/1400/1*neBcAZJyLGpE7KHc3sH8bw.png, August 2021. Accessed: 08/02/2022.

[18] Sankalp Jonna. Timeloop. https://github.com/sankalpjonn/timeloop, 2020. Accessed: 08/02/2022.

[19] STUDIO SAMO. Kpi. https://www.studiosamo.it/glossario/kpi/. Accessed: 10/02/2022.

[20] Apache Software Foundation. Apache kafka. https://kafka.apache.org/, .

[21] MongoDB Inc. Mongodb. https://www.mongodb.com/.

[22] Vineet Goel Ask Solem. Faust. https://github.com/robinhood/faust.

[23] RedHat. Come funziona apache kafka? https://www.redhat.com/it/topics/integration/what-is-apache-kafka, October 2017. Accessed: 12/02/2022.

[24] TIBCO. Cos'è apache kafka? https://www.tibco.com/it/reference-center/what-is-apache-kafka. Accessed: 12/02/2022.

[25] Markus Gulden. Introduction to kafka connectos. https://www.baeldung.com/kafka-connectors-guide, August 2021. Accessed: 12/02/2022.

[26] <html>.it. Guida mongodb - introduzione. https://www.html.it/pag/52182/introduzione-15/. Accessed: 12/02/2022.

[27] Apache Software Foundation. Kafka streams. https://kafka.apache.org/0102/documentation/streams/, . Accessed: 12/02/2022.

[28] Robinhood Markets. Faust - python stream processing. https://faust.readthedocs.io/en/latest/, . Accessed: 12/02/2022.

[29] Bitnami. Apache zookeeper packaged by bitnami. https://hub.docker.com/r/bitnami/zookeeper/, . Accessed: 12/02/2022.

[30] Bitnami. Apache kafka packaged by bitnami. https://hub.docker.com/r/bitnami/kafka/, . Accessed: 12/02/2022.

[31] obsidiandynamics. Kafdrop docker image. https://hub.docker.com/r/obsidiandynamics/kafdrop. Accessed: 12/02/2022.

[32] confluentinc. Confluent docker image for the rest proxy. https://hub.docker.com/r/confluentinc/cp-kafka-rest. Accessed: 12/02/2022.

[33] Debezium. What is debezium? https://debezium.io/documentation/reference/1.8/, . Accessed: 13/02/2022.

[34] l'enciclopedia libera Wikipedia. Change data capture. https://en.wikipedia.org/wiki/Change_data_capture. Accessed: 13/02/2022.

[35] Debezium. Debezium connector for mongodb. https://debezium.io/documentation/reference/stable/connectors/mongodb.html, . Accessed: 13/02/2022.

[36] Debezium. Debezium zookeeper. https://hub.docker.com/r/debezium/zookeeper, . Accessed: 15/02/2022.

[37] debezium. Debezium kafka. https://hub.docker.com/r/debezium/kafka. Accessed: 15/02/2022.

[38] Debezium. Debezium connect. https://hub.docker.com/r/debezium/connect. Accessed: 15/02/2022.

[39] Confluent. Connect rest interface. https://docs.confluent.io/platform/current/connect/references/restapi.html#. Accessed: 15/02/2022.

[40] Robinhood Markets. Faust - tables and windowing. https://faust.readthedocs.io/en/latest/userguide/tables.html, . Accessed: 17/02/2022.

[41] W3C Working Group. Json-ld 1.1 - a json-based serialization for linked data. https://www.w3.org/TR/json-ld11/#basic-concepts, July 2020. Accessed: 19/02/2022.

[42] Maximilian Ventura. Linked data. https://fontistoriche.org/wp-content/uploads/linked-data-1024x386.png, December 2020. Accessed: 19/02/2022.

[43] W3C Working Groupa. Curie syntax 1.0. https://www.w3.org/TR/curie/, December 2010. Accessed: 19/02/2022.

[44] W3C Working Groupa. https://www.w3.org/tr/shacl/. https://www.w3.org/TR/shacl/, July 2017. Accessed: 20/02/2022.

Ringraziamenti

Un grazie doveroso e sincero al mio relatore, il professore Giovanni Malnati, per tutto ciò che mi ha insegnato sia a livello tecnico sia soprattutto a livello umano.

Desidero ringraziare altrettanto sinceramente il mio correlatore, il professore Fabio Forno, per essere stato un costante punto di riferimento durante lo sviluppo di questa tesi.

Grazie a Matilde, per la gentilezza e la disponibilità dimostratami durante lo sviluppo della mia tesi.

Grazie a Vincenzo, per essere con me fin dai tempi delle superiori e per essere stato il primo a credere che in me c'è del potenziale.

Grazie a Marco, per essere per me un caro amico di cui potermi fidare ciecamente in ogni momento e per tutto ciò che abbiamo condiviso insieme sin dal mio primo anno di università.

Grazie a Giorgia, che nonostante i suoi millanta impegni ha fatto sempre il possibile per trovare del tempo da condividere con me sin dal mio primo anno di università.

Grazie a tutti i membri del mio team Rattlesmake: Fabio, Giorgio, Gaetano, Claudio e Marjo, per essere vicini a me da anni - condividendo gioie, problemi e passioni - e per lavorare insieme alla realizzazione di un grande sogno in comune per il nostro futuro e per il supporto che ci diamo l'uno con l'altro per la nostra crescita personale.

Grazie a Emanuele e Giulia, per avermi insegnato che bisogna sempre provare a sorridere, anche nelle più grandi difficoltà.

Grazie a Francesco C, per essere un amico con cui poter parlare di ogni cosa e per essere stata la persona più presente nel momento finale, e per certi versi critico, della mia tesi.

Grazie a tutte le persone che mi hanno tenuto compagnia in residenza universitaria, nell'anno di scrittura della mia tesi. In particolare grazie a Vittoria, Carla, Francesco G, Antonio, Ismaele, Chiara, Nicola, Matteo, Valeria, Ainura, Daniele, Tito e Umar.

Grazie poi a tutte le persone che ho conosciuto in 6 anni di residenza universitaria per i bei momenti trascorsi; in particolare grazie a Emanuela, Sabrina, Antonina, Lorenzo, Clarissa, Eugen, Giacomo, Valentina, Mounif, Alberto, Erika, Gianpiero, Gianluca, Luca, Cosmin e Ludovica.

Infine, ringrazio con tutto il cuore i miei genitori e mia sorella, che da sempre credono in me, mi amano e mi sostengono.