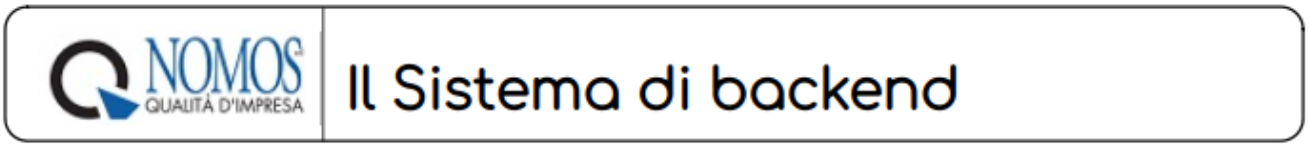

Darm-Backend

Documentazione tecnica

Version v.1.1

Sommario

Chapter 1. La situazione attuale

La piattaforma ICA-Darm deve essere alimentata da flussi mensili relativi ai consumi farmaceutici per centro di costo (ddd, protesi, cvc, gel) e alle giornate di degenza in regime di ricovero ordinario. La funzione del backend è di acquisire, validare e caricare su database i dati dei flussi mensili per renderli accessibili attraverso viste dedicate al frontend della piattaforma.

Da una situazione iniziale improntata a una gestione esclusivamente manuale, in previsione di un crescente numero di aziende utenti della piattaforma, il backend è stato oggetto di una progressiva evoluzione che ha portato a una gestione automatizzata degli *uploads* e delle relative notifiche, attraverso un'architettura rappresentata nella figura che segue.

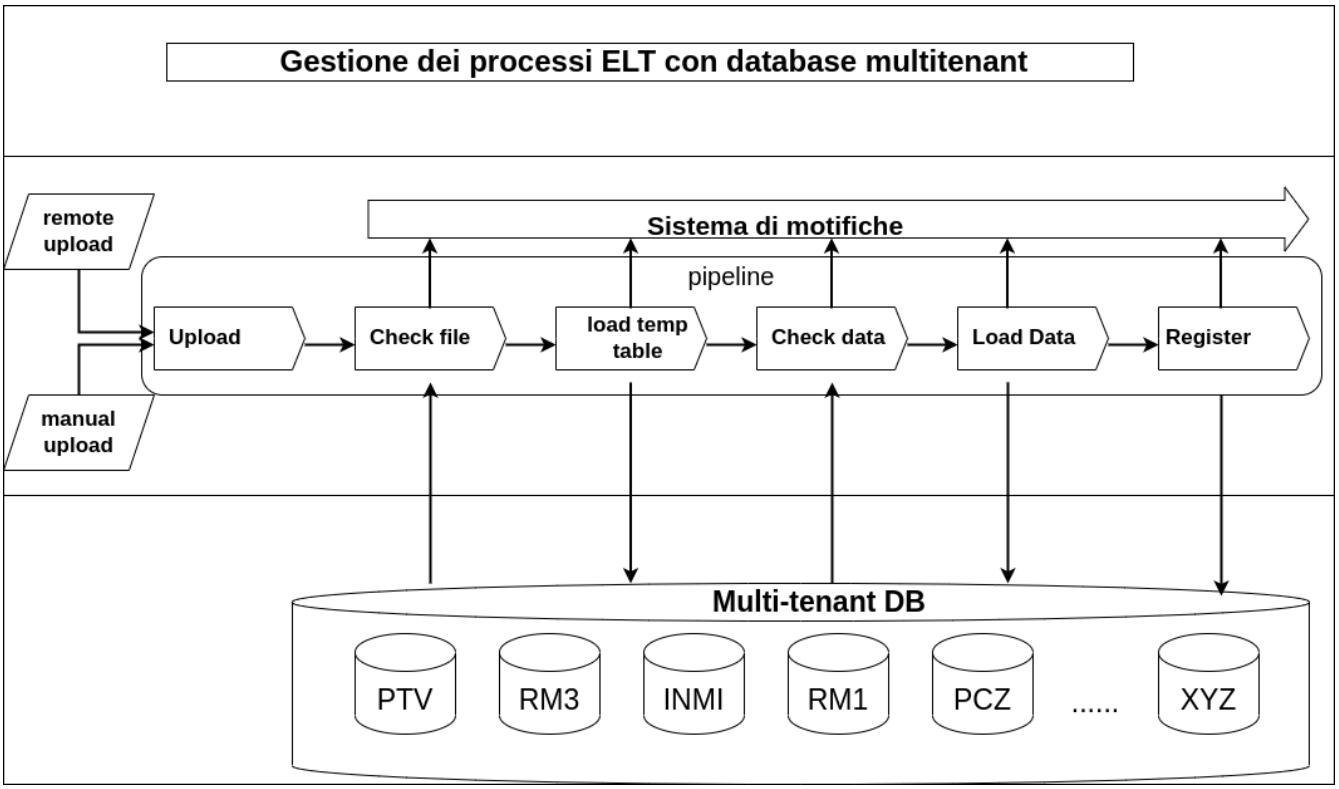

Fig. 1 - Architettura del backend

L'architettura del sistema attuale si articola in due principali componenti:

- l'applicazione Go, che:
  - fornisce le interfacce per le differenti modalità di upload
  - gestisce attraverso la *pipeline* la *business logic* dei processi di caricamento dei dati [1]
  - fornisce un sistema di notifiche in tempo reale per l'aggiornamento degli utenti sui processi di caricamento
  - consente la tracciabilità dei processi di caricamento
- il database Postgresql, che:
  - riserva uno schema dedicato per ogni azienda e per i relativi flussi di dati,

- implementa le tabelle di destinazione dei dati e le viste per il frontend della piattaforma
- supporta gli *steps* della *pipeline* attraverso funzioni generiche che si applicano a tutti gli schemi
- ha una potenziale scalabilità in termini di aziende gestibili che supera abbondantemente il numero totale delle aziende sanitarie in Italia [2]

Nei paragrafi che seguono si approfondiscono le componenti e le funzionalità sintetizzate nei punti precedenti.

[1] si usa il termine generico "processi di caricamento" evitando il ricorso a "ETL" perché questa sigla rappresenta solo in parte tali processi e l'altra parte di fatto prevalente è l'"ELT" in cui le trasformazioni dei dati sono effettuate dopo il caricamento nel database

[2] Alcuni *posts* su StackOverflow fanno riferimento a casi in cui l'organizzazione per schemi di Postgresql consente di gestire oltre i 100.000 tenants, con l'unico downside in termini di performance rappresentato dalla lentezza dei backups

Chapter 2. L'applicazione GO

2.1. La configurazione dei flussi e delle aziende

La configurazione dei flussi e delle aziende è la base dell'intera applicazione ed è stata impostata per facilitare modifiche che non richiedano interventi sul *codebase*. In questo modo, aggiungere una nuova azienda o un nuovo flusso o modificare un flusso esistente diventa un'operazione semplice, basata sulla conoscenza dei metadati su cui si basano sia l'applicazione Go che il database. Per contro, l'implementazione a livello di database di queste modifiche non può prescindere, attualmente, da una competenza specialistica in SQL per creare o modificare oggetti del database (tabelle, viste, schemi, etc), garantendo al tempo stesso l'integrità dei dati e la coerenza del sistema.

Rinviando ad una sezione dedicata l'aggiunta di una nuova azienda, **di seguito si descrive la configurazione dei flussi e delle aziende limitatamente all'applicazione in Go**.

La configurazione del singolo flusso viene specificata attraverso un file *yaml* come nell'esempio che segue:

Esempio di configurazione del flusso ddd_consumi

```
     ddd_consumi:                        # il flusso
        schema: ptv                      # lo schema del database
        target_table: ddd_consumi        # il nome della tabella in cui sono
registrati i dati
        columns: 12                      # numero di colonne previste per il file csv
        filename: ptv_ddd_consumi_AAMM.csv # template del file name
        delimiter: "|"                    # delimitatore di campo
        header:
cdc|descr_cdc|atc|descr_atc|cod_prodotto|descr_prodotto|um|quantita|ddd|ddd_um|valore|
anno_mese # in quale                                  #ordine devono essere le colonne
        archive_folder: ddd              # cartella in cui sono archiviati i files
caricati
        scadenza_mensile: 5              # giorno del mese in cui è previsto il
caricamento dei dati
        user: xxx.yyy@ptvonline.it       # il referente aziendale responsabile del
caricamento
```

Il livello superiore al flusso è l'azienda che invia i dati, in questo caso PTV, e che a sua volta viene descritta come un insieme di flussi, come nell'esempio che segue:

Esempio di configurazione di un'azienda

```
    ptv:
      ddd_consumi:                       # il flusso ddd
        filename: ptv_ddd_consumi_AAMM.csv
```

```
      .....
    cvc_consumi:                        # il flusso cvc
      filename: ptv_cvc_consumi_AAMM.csv
      .....
    gel_consumi:                        # il flusso gel
      filename: ptv_gel_consumi_AAMM.csv
      .....
    pro_consumi:                        # il flusso protesi
      filename: ptv_pro_consumi_AAMM.csv
      .....
    sio_adt:                            # il flusso sio adt
      filename: ptv_sio_adt_AAAAMMGG.zip
      .....
    sio_sdo:                            # il flusso sio sdo
      filename: ptv_sio_sdo_AAMM.zip
      .....
```

Estendendo questa logica, l'insieme delle aziende e dei flussi sono contenuti nel file *flows.yml* mentre in un altro file *config.yml* altri aspetti si aggiungono alla descrizione di un'azienda.

I descrittori di un'azienda

```
aziende:
  ptv:
    descrizione: "Policlinico Tor Vergata"
    db_schema: ptv
    cdc_length: 12
    flussi_mensili: [ddd_consumi,cvc_consumi,pro_consumi,gel_consumi,sio_adt]
  inmi:
    descrizione: "INMI Lazzaro Spallanzani"
    db_schema: inmi
    cdc_length: 10
    flussi_mensili: [ddd_consumi,cvc_consumi,gel_consumi,sio_adt]
```

Da notare in questa struttura **il campo dei flussi mensili che elenca i flussi da includere nelle *chiusure mensili***, ovvero quali flussi costituiscono il debito informativo mensile di ogni azienda. Questa aggiunta è importante a supporto dell'*operation management* dei processi di caricamento che deve garantire alla piattaforma tempistiche costanti nella consegna mensile dei flussi.

2.2. Altri parametri di configurazione

Oltre ai metadati relativi ai flussi e alle aziende, esistono altri parametri di configurazione dell'applicazione Go, alcuni dei quali contenuti nel già citato *config.yml* e che riguardano:

```
dbpool:
```

```
db_max_open_conns: 100
db_max_idle_conns: 26
db_max_idle_time: "15m"
port: 4000
firebase_sdk: pkg/config/firebase-adminsdk.json
```

Infine, il file *.env* contiene dati sensibili relativi a:

- la connessione al database;
- l'accesso a Firebase per la gestione utenti;
- la *secret key* per la firma dei token JWT.

2.2.1. La naming convention

Per la configurazione del sistema di controllo sono indispensabili i *metadati*, cioè dati che descrivono altri dati, in questo caso flussi e aziende, e che sono utilizzati per organizzare, individuare e gestire i dati stessi. La *naming convention* è un caso esemplare di metadato. Considerato che tutto il sistema di backend si basa sui *files* di input, il nome dei files non può essere lasciato al caso o alla discrezionalità del referente aziendale, ma deve essere standardizzato attraverso una convenzione che ne garantisca l'identificazione univoca e la corretta collocazione nel database e nel filesystem.

azienda	***Nome flusso***	***periodo***	***estenione***
ptv	sio_sdo	(20)2310	csv/zip

Questa segmentazione del nome del file si rivela utile sotto vari profili:

- la mappatura sul filesystem, ovvero la creazione di cartelle organizzate per azienda/flusso in cui vengono archiviati i *files* utilizzati per i caricamenti;
- la mappatura sul database, dove consente di individuare in quale schema e in quale tabella dovranno essere caricati i dati;
- la mappatura sugli *endpoints*, in cui la *url* può essere composta dai segmenti della *naming convention*.

2.2.2. Altre specifiche di configurazione dei flussi

Ci sono altri campi nella configurazione dei flussi che devono essere menzionati, quali:

- **il numero di colonne del file csv** che per motivi diversi potrebbe variare rispetto allo standard, per esempio perché il delimitatore di colonna '|' è utilizzato in un campo descrittivo oppure perché nell'excel utilizzato per l'esportazione in csv è rimasta una colonna di troppo, due casi che si sono verificati nella pratica e in aziende diverse;

- **l'elenco dei campi** che deve essere presente nella prima riga del file csv, per evitare che un file con i campi invertiti venga caricato nel database, come è successo in un caso reale relativo al gel idroalcolico quando ad un certo punto sono state invertite le colonne della quantità e del valore (che nel database hanno lo stesso tipo di dato *numeric*), con il risultato che per mesi la tabella dei consumi di gel ha riportato dati sbagliati e li ha passati alla piattaforma di frontend;
- **la lunghezza del codice dei centri di costo finali**, utile per evitare il caricamento di dati aggregati, come si è verificato per i consumi DDD per i quali sono stati acquisiti alcune volte solo i dati aggregati a livello dipartimentale, quindi inutilizzabili per la costruzione dell'indicatore DDD/giornate di degenza che deve essere implementato sui centri di costo finali;
- un eventuale **postprocess**, ad esempio per il refresh di viste materializzate o per altre operazioni specifiche del flusso.

2.3. Configurazione della struttura nel database

Nel database la configurazione della struttura è descritta attraverso schemi e tabelle come evidenzia la figura seguente.

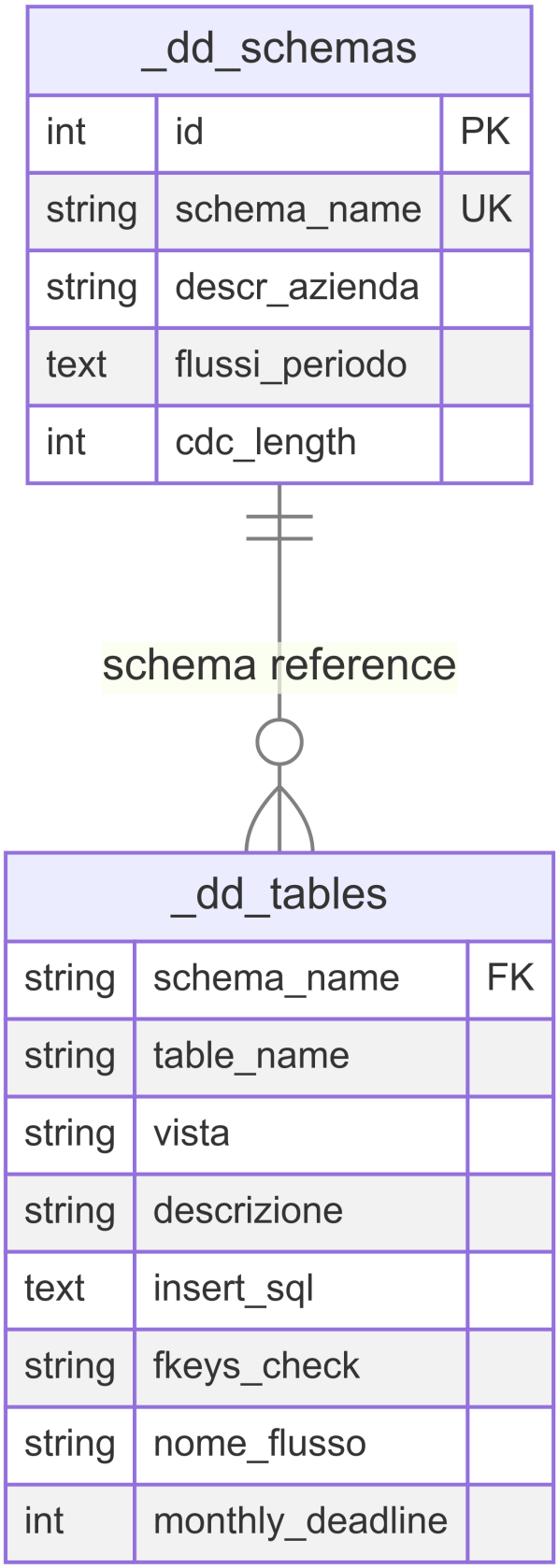

Fig. 2 - Metadati - La struttura aziendale

Ogni schema corrisponde ad un'azienda e per ogni schema ci sono le tabelle target e ogni tabella corrisponde ad un flusso.

Da sottolineare:

- nella tabella *dd_schemas* il campo flussi riporta l'elenco dei flussi che ogni mese devono essere aggiornati per l'alimentazione della piattaforma. Tale elenco può differire dai flussi presenti in *_dd_tables*, come nel caso del SIO al PTV che non è più utilizzato per i dati relativi alle giornate di degenza ma solo per gli indicatori. Per definire in modo chiaro quali flussi sono da considerare per l'alimentazione della piattaforma nella tabella *dd_schemas* è stato introdotto il campo *flussi_periodo* il che consente poi di definire le chiusure mensili come dato di sintesi ai fini del monitoraggio dei processi di caricamento;
- nella tabella *dd_tables* è indicata la scadenza mensile per il caricamento dei dati, che è un parametro importante per l'operational management dei processi di caricamento.

2.4. La standardizzazione dell'input

Le modalità di configurazione dei flussi e delle aziende sopra descritte rientrano in una più ampia strategia di standardizzazione dell'input come requisito per l'automazione dei processi di caricamento dei dati e per minimizzare i costi di manutenzione del sistema, in particolare quando si tratta di integrare una nuova azienda.

Il documento relativo ai requisiti dei flussi dei consumi e dell'attività, che è stato oggetto di una prima stesura nel 2022 ed è tutt'oggi utilizzato in fase di acquisizione di nuove aziende, presenta costantemente gli stessi tracciati record per le tabelle dei consumi di DDD, di protesi, etc. il che non preclude la possibilità che vengano apportate variazioni giustificate in base al contesto locale.

2.5. Modalità di upload

2.5.1. Caricamento manuale

L'interfaccia utente è stata implementata attraverso una form che:

- consente di selezionare il file da caricare;
- riporta nome e dimensioni del file;
- riporta l'identificativo assegnato al processo, che accompagna tutta la *life cycle* della pipeline;
- rappresenta visivamente la sequenza degli *steps* della pipeline attravero 5 forme su cui lo stato di avanzamento del processo è visualizzato con una progressione di colori;
- riporta all'*uploader* l'esito di ogni fase con un messaggio che può includere la descrizione di eventuali errori segnalati dalla pipeline.

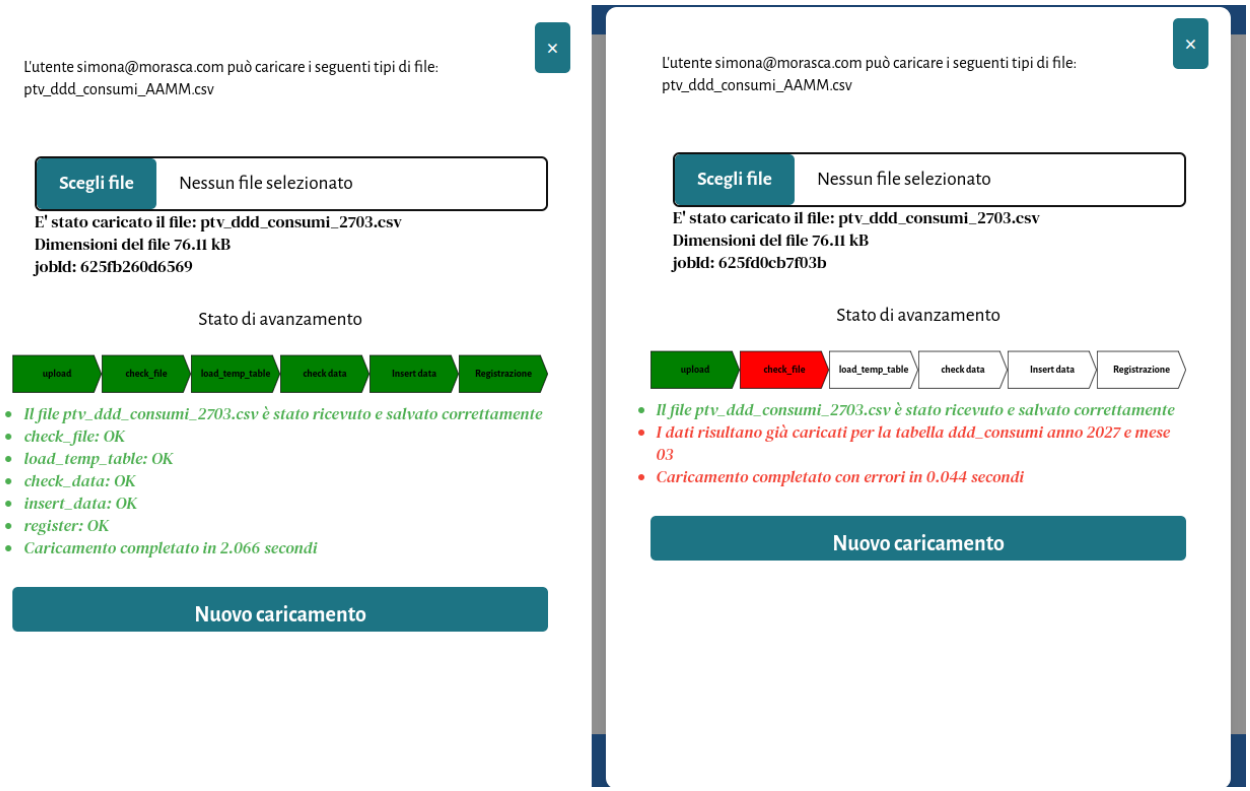

Fig. 3 - Notifiche di successo e di errore

La risposta sull'esito del caricamento, come sopra evidenziato, non viene fornita al termine dell'operazione ma con aggiornamenti in tempo reale che sono gestiti dal sistema di notifiche più avanti specificato.

2.5.2. Remote Upload

Il *remote upload* non è stato ancora attivato ma, tenuto conto che è prevista una forte crescita di questa modalità è stato sviluppato e testato un *client* in Go con interfaccia CLI. Il *client* si integra perfettamente con la pipeline e con il sistema di notifiche.

L'interfaccia utente, in questo caso, è ovviamente molto diversa dal manual upload, come evidenzia lo screenshot di seguito riportato.

```
mm@thp-p15...ione/18-10#>./rest_post -h
[redacted] INF Loaded 187 requests
Usage of ./rest_post:
  -baseUrl string
        Base URL for the requests (required)
  -numRequests int
        Number of requests to post (default 1)
  -useWorkpool
        Use a work pool for requests (default false)
  -chooseFile
        Indica il nome del file da postare
```

Fig. 4 - Interfaccia utente del client per il remote upload

Si può selezionare un file specifico, oppure si può specificare il numero delle richieste da lanciare.

Anche la presentazione degli *outputs* - almeno nel prototipo sviluppato - è diversa e si limita al flusso su STDOUT, ma, come si può vedere nello screenshot seguente, è possibile visualizzare lo stato di avanzamento del processo e ricevere eventuali messaggi di errore.

```
INF PostRequest 45  file=ptv_pro_consumi_2703.csv user=obiwankenobi@gmail.com
INF Status 200 OK: Response data: Il file ptv_pro_consumi_2703.csv è stato ricevuto e salvato corrett
onsumi_2703.csv jobId=626b423a943cd user=obiwankenobi@gmail.com
INF Response data: check_file: OK file=ptv_pro_consumi_2703.csv jobId=626b423a943cd user=obiwankenobi

INF Response data: load_temp_table: OK file=ptv_pro_consumi_2703.csv jobId=626b423a943cd user=obiwank

INF Response data: check_data: OK file=ptv_pro_consumi_2703.csv jobId=626b423a943cd user=obiwankenobi

INF Response data: insert_data: OK file=ptv_pro_consumi_2703.csv jobId=626b423a943cd user=obiwankenob

INF Response data: register: OK file=ptv_pro_consumi_2703.csv jobId=626b423a943cd user=obiwankenobi@g

INF pipeline completed file=ptv_pro_consumi_2703.csv jobId=626b423a943cd user=obiwankenobi@gmail.com
```

Fig. 5 - Output del client per il remote upload

2.5.3. Autenticazione e autorizzazione

L'autenticazione viene gestita con due differenti modalità:

- per gli utenti *uploader* l'autenticazione è gestita attraverso **firebase**, utilizzando un account intestato `ica-backend`, e attraverso un apposito middleware che esamina ogni richiesta;
- per i providers di *remote upload* l'autenticazione è gestita attraverso un JWT generato dal server e verificato ad ogni richiesta da un middleware dedicato. Ovviamente questa soluzione è una scelta di implementazione che non preclude scenari diversi.

A supporto dei due sistemi di autenticazione ci sono due tabelle nel database: *admin.users* e

admin.providers.

Per quanto riguarda l'**autorizzazione**, ci sono due livelli:

- il ruolo di *uploader* assegnato da un amministratore Nomos in fase di creazione utente;
- quali files l'*uploader* o il *provider* possono caricare, attraverso un elenco di tipi di file memorizzato nel record utente o provider.

Nel caso dell'uploader, la form di caricamento visualizza in prima riga il dettaglio dei file templates che l'utente può caricare. Al tentativo di caricare un file non riconducibile all'elenco, sia per il manual che per il remote upload, il server risponde con un messaggio di errore:

```
Error uploading file: upload failed with status: 403 Forbidden, message: Non ha i
permessi per caricare questo tipo di file
```

2.6. La pipeline

Cosa accade quando un file viene caricato nel sistema? Secondo una logica *producer-consumer*, l'upload produce dei dati che sono "consumati" dal processo successivo. Il prodotto dell'upload è un insieme di metadati basati sulla *naming convention* che confluiscono in una *FlowInstance*, cioè la struttura dati che verrà consumata/trasformata dalla pipeline. Il metadato principale prodotto dall'upload è il *jobId* che identifica la *FlowInstance* e ne consente la tracciabilità.

La pipeline è **un insieme di passaggi che vengono eseguiti in sequenza per completare il processo di caricamento e in questi passaggi la *FlowInstance* viene modificata in relazione allo stato di avanzamento**. Tale sequenza è rappresentata nella figura seguente.

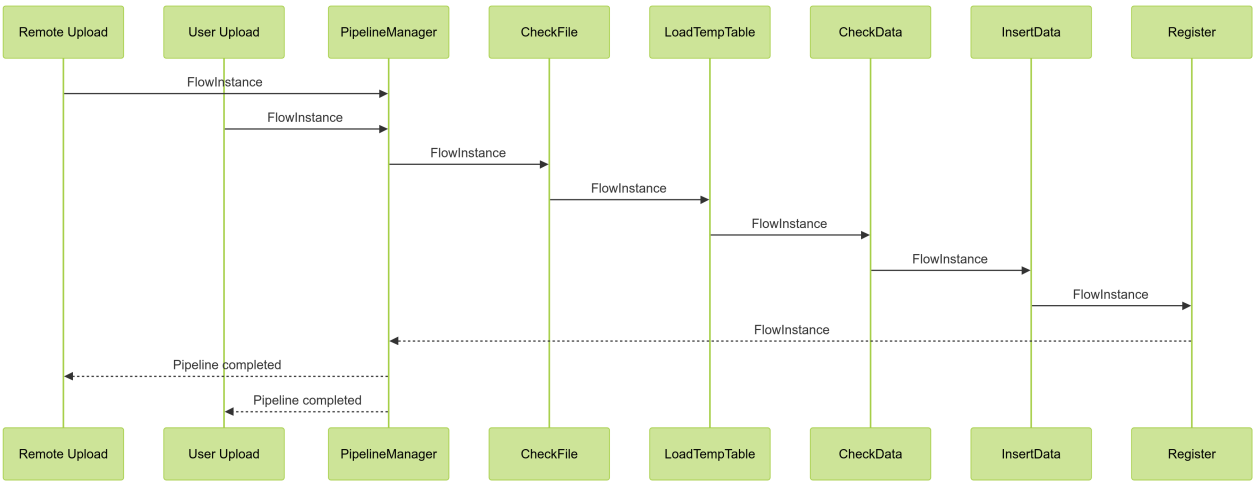

Fig. 6 - La pipeline

Di seguito si descrivono i passaggi principali.

2.6.1. Upload

Il caricamento del file può avvenire in due modi: manualmente, da parte di un utente che attraverso il browser sceglie il file da caricare, o attraverso un servizio di upload remoto. In entrambi i casi esistono procedure di **autenticazione e autorizzazione** che salvaguardano la sicurezza del sistema di controllo.

A prescindere dalla modalità di caricamento, manuale o remoto, la verifica preliminare riguarda la *naming convention* e in caso di esito positivo:

- il file viene caricato e salvato in una cartella temporanea;
- viene generata una nuova *FlowInstance*.

La flowInstance

Come già precisato in tema di *naming convention*, il nome del file consente di identificare l'azienda e il flusso di riferimento con tutti i metadati associati. Quelle informazioni sono integrate dal periodo di riferimento dei dati, contenuto nel nome del file. In base all'insieme di queste informazioni viene attivata una **flowInstance** che, come evidenziato in fig. 2, viene passata da uno stadio all'altro della *pipeline*. La *flowInstance* viene associata ad un *jobId*, un identificativo univoco utilizzato per la tracciabilità del processo.

Nella tabella seguente sono riportati i campi della *flowInstance* e la descrizione del relativo contenuto.

Nome del campo dati	Contenuto del campo dati
Db *pgxpool.Pool	database connection pool
Log zerolog.Logger	logger
FlowName	nome del flusso dati
FilenameTemplate	naming convention
ArchiveFolder	cartella di archiviazione
DatafilesRoot	radice della cartella di archiviazione
Filepath	percorso completo del file
SchemaName	schema del database = sigla azienda
TableName	nome della tabella di destinazione
Columns	numero di colonne
Delimiter	delimitatore di colonna
Header	prima riga del csv con nomi dei campi
FieldNames []string	nomi dei campi nella tabella su db
nome della tabella temporanea su db	TempTable
Anno	anno di competenza dei dati

Nome del campo dati	Contenuto del campo dati
Mese	mese di competenza dei dati
Md5sum	identificazione univoca del file nel db e nel filesystem
Status	status del file (unloadable,loadable,registered,loaded)
CsvLines	numero di righe del file csv
LoadedLines	numero di righe caricate nella tabella finale
LoadingMethod	metodo di caricamento (manuale o remoto)
JobId	identificativo del processo utilizzato per il tracking e il logging
User	utente che ha effettuato l'upload

2.6.2. Pipeline start

La *FlowInstance* viene sottoposta al *PipelineManager* attraverso l'endpoint */api/rest/start-process/{jobId}*. Il *PipelineManager* dispone di un pool di *goroutines* tramite le quali gestisce richieste concorrenti di attivazione di pipeline. Di seguito si descrivono i vari passaggi della pipeline.

2.6.3. CheckFile

CheckFile è la funzione che esegue le prime verifiche sul file appena caricato, che riguardano:

- un controllo preliminare sul database per verificare se i dati siano già stati caricati;
- il mime-type, per confermare che si tratta di un file "text/plain";
- il delimitatore di colonna e il numero di colonne previste;
- il formato della data o del periodo di riferimento dei dati.

In caso di errori la pipeline viene interrotta e il tipo di errore viene segnalato all'utente che ha iniziato l'upload. Se le verifiche hanno esito positivo la *flowinstance* viene integrata con:

- il numero delle righe;
- la md5sum, che identifica univocamente il file.

La pipeline avanza allo step successivo.

2.6.4. LoadTempTable

In questa fase viene utilizzata la tabella temporanea che corrisponde al flusso di riferimento del file. La tabella temporanea è una tabella di appoggio con una struttura sovrapponibile alla tabella

target e che ha due funzioni distinte:

- caricare su campi in formato testo dati che sono in formato numerico o data e che non possono essere caricati direttamente nella tabella target;
- verificare la correttezza dei dati prima del caricamento nella tabella target.

Il caricamento dei dati nella tabella temporanea è eseguito in due *steps*:

- vengono cancellati tutti i dati eventualmente presenti nella tabella;
- viene eseguito il comando COPY per l'importazione massiva dei dati.

2.6.5. CheckData

CheckData è la funzione che esegue la validazione dei dati caricati nella tabella temporanea e in caso di esito negativo gestisce le eventuali violazioni ai *constraints* esistenti.

Le verifiche riguardano in particolare i *foreign key constraints* che garantiscono l'integrità referenziale tra le tabelle (es. il codice reparto che si trova in un caso di ricovero deve essere presente nella tabella dei reparti, il codice centro di costo nei consumi DDD deve essere presente nella tabella dei centri di costo, etc.).

La funzione esegue i seguenti passaggi:

1. controlla le chiavi esterne mancanti utilizzando la funzione di database *check_fks* che prende come parametri *schema_name*, *table_name*, *jobId*;
2. Se vengono trovate chiavi mancanti, le inserisce nella tabella *_dd_missing_fkeys*, specificando il tipo di chiave mancante (es. reparto, centro di costo, etc.), il valore mancante, il job_id e la data di inserimento, da utilizzare per la tracciabilità del problema e per la comunicazione con l'azienda interessata;
3. Per ogni chiave esterna mancante, tenta di inserire i valori mancanti nella tabella esterna corrispondente, che dovrà comunque essere integrata con le relative descrizioni che saranno fornite dall'azienda interessata. L'aggiunta di questo step è stata introdotta per evitare che il processo di caricamento si interrompa per una chiave esterna mancante, con un conseguente ritardo di giorni persi nella comunicazione del problema e nell'attesa della soluzione.

2.6.6. InsertData

La funzione inserisce i dati della tabella temporanea nella tabella di destinazione, con i seguenti steps:

1. L'inserimento dei dati viene effettuato utilizzando la funzione del databse *insert_from_temp_table* che prende come parametri *schema_name* e *target_table_name*;
2. La funzione del database ritorna una stringa con due valori separati dal delimitatore '|' e i seguenti possibili risultati:

    - se ci sono stati errori il primo valore è -1, mentre il secondo valore riporta il codice di errore;
    - se l'inserimento è avvenuto correttamente il primo valore è il numero di righe inserite;
3. se l'inserimento è avvenuto correttamente, la funzione aggiorna i campi *LoadedLines* e *Status* di *FlowInstance*;
4. esegue eventuali istruzioni SQL di post-elaborazione se specificate (ad es. refresh di viste materializzate, etc.).

La funzione gestisce diversi tipi di errore:

- se non vengono inserite righe, restituisce un **errore di dominio**;
- se c'è un errore durante l'inserimento, restituisce un **errore di sistema**.

2.6.7. Register

È l'ultima fase della pipeline in cui si esegue la registrazione nel database dell'avvenuto caricamento. In particolare sono registrati i seguenti dati:

- il nome dello schema del database;
- il nome del flusso dati;
- il nome della tabella di destinazione;
- il percorso completo del file;
- la md5sum del file;
- l'anno e il mese di competenza dei dati;
- il numero di righe del file csv;
- il numero di righe caricate nella tabella target;
- il job_id;
- l'utente che ha effettuato l'upload;
- il metodo di caricamento (manuale o remoto);
- la data e l'ora dell'operazione;
- lo stato del file (unloadable,loadable,registered,loaded).

Questi dati sono registrati nella tabella *dd_csv_files_ che è collegata alla tabella* dd_tables_ come nel diagramma che segue.

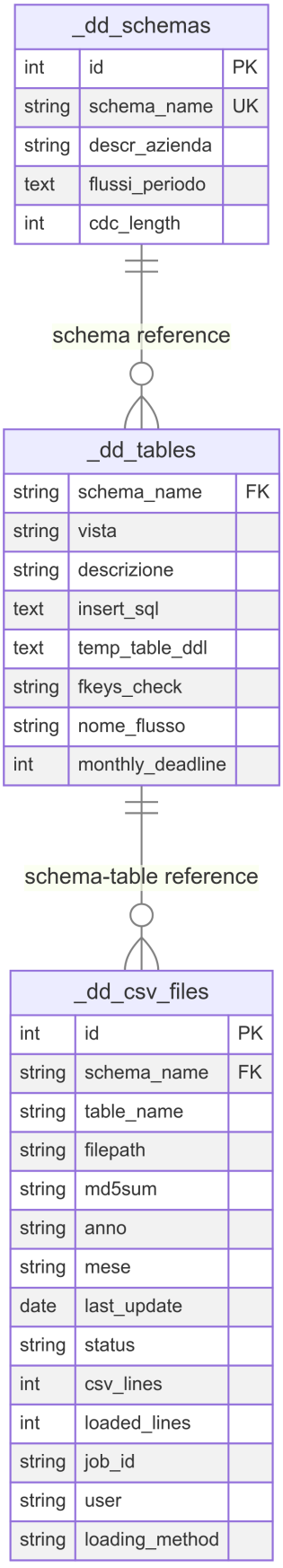

Fig. 7 - Le tabelle dd_csv_files e dd_tables

Questa tabella è importante perché:

- è accessibile nella dashboard all'utente *uploader* che trova un riscontro immediato sul risultato finale del caricamento;
- ai fini dell'*operational management* è la base del **sistema di monitoraggio** per sapere se tutti i caricamenti previsti sono stati effettuati, per confrontare le scadenze mensili in *_dd_tables* con le date di effettivo caricamento, etc.

Le chiusure mensili

Una domanda frequente è: "a che punto siamo con i caricamenti mensili dell'azienda XYZ?" Tutti i flussi sono stati caricati?".

Per rispondere a questa domanda è stata predisposta una vista che riporta i flussi mensili di ogni azienda e il loro stato di avanzamento; tale vista è disponibile nella dashboard dell'utente *uploader*, limitatamente all'azienda di appartenenza, e nella dashboard dell'utente *admin*. Questa soluzione non è priva di limiti: richiede un accesso a *darm-backend*, deve essere interpretata (perché nello Spallanzani la colonna *pro* è piena di 0? perché nel PTV non è riportato il flusso SIO?) e, soprattutto, non è oggetto di alcuna notifica ai referenti aziendali, ai fornitori della piattaforma e al management Nomos.

In considerazione di questi limiti, sono state introdotte le *chiusure mensili*. Nella tabella *dd_csv_files_ è stato installato un trigger che ad ogni caricamento confronta i caricamenti effettuati con i caricamenti attesi, riportati nella tabella* dd_schemas_. Se tutti i caricamenti sono stati completati, la chiusura mensile è registrata in un'apposita tabella che integra il modello dei dati come nel diagramma di seguito riportato. Questa tabella a sua volta potrebbe essere una fonte di notifiche per le aziende, per i fornitori della piattaforma e per il management Nomos.

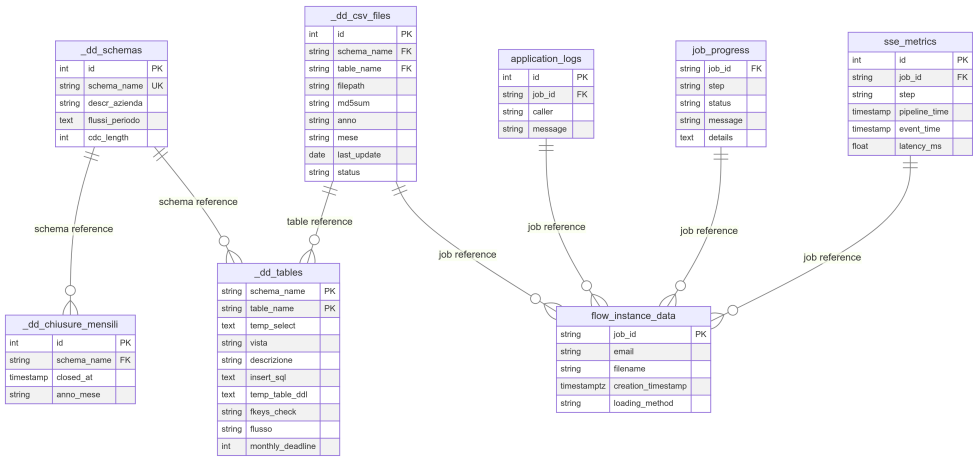

Fig. 8 - Integrazione del modello dei dati con le chiusure mensili

Nel momento in cui risulta completato l'aggiornamento dei flussi mensili, viene aggiunto un record alla tabella *_dd_chiusure_mensili* che specifica l'azienda, la data e il periodo Anno-Mese a cui si riferisce la chiusura. Inoltre un *trigger* sulla stessa tabella innesca la produzione di riepiloghi con dati aggregati in excel, che gli utenti possono scaricare per verificare i dati caricati sui quali sono

costruiti gli indicatori OMS.

Inoltre, per il flusso ADT è disponibile anche un report per singolo paziente con il dettaglio degli eventi relativi al ricovero (accettazione, eventuali trasferimenti, e dimissione) e delle unità operative che hanno ospitato il paziente.

2.7. Il sistema di notifiche

La pipeline aggiorna continuamente lo stato di avanzamento delle *FlowInstance* attive e questi aggiornamenti sono registrati nella tabella *admin.job_progress* su cui si appoggia il sistema di notifiche. In questa tabella, come si può notare nel diagramma che segue, ogni record riporta lo stato della pipeline relativa ad uno specifico processo di caricamento.

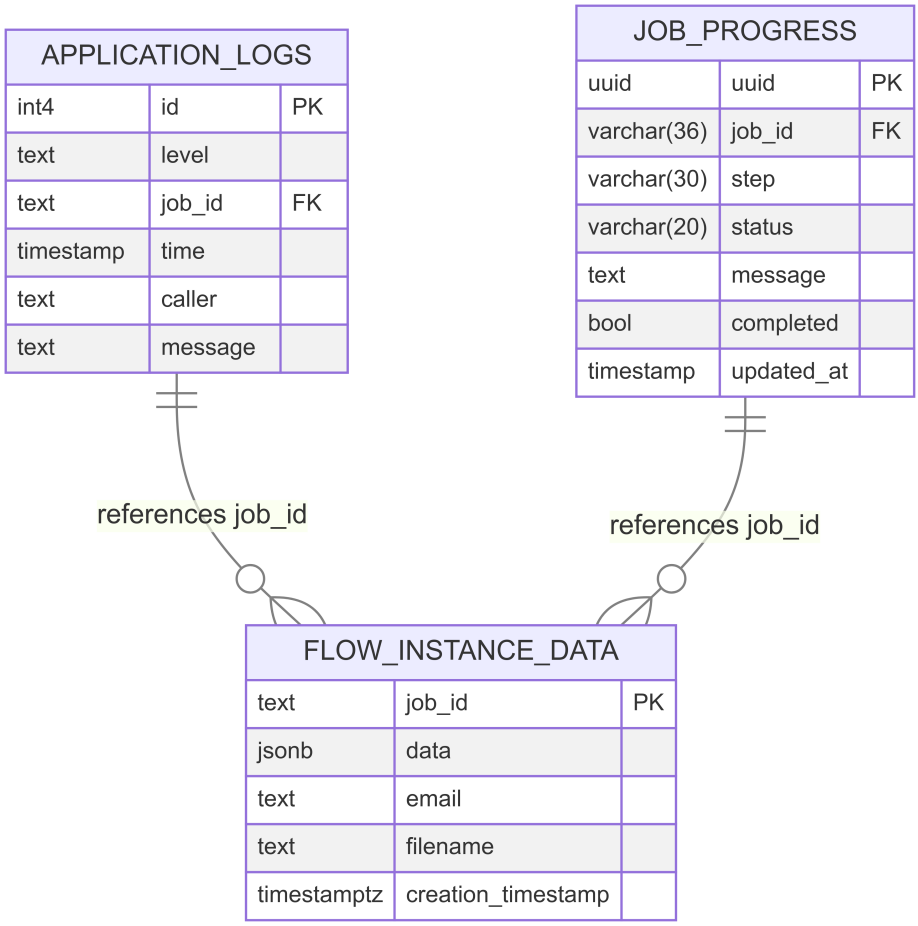

Fig. 9 - Stato di avanzamento della pipeline

Il sistema di notifiche trasferisce questi aggiornamenti in tempo reale agli utenti utilizzando una logica *pull* o una logica *push*. Nel primo caso è il client che richiede lo stato di avanzamento attraverso una sequenza di GET, nel secondo caso è il server che, ad esempio attraverso il **Server-Sent-Events**, attiva lo streaming delle notifiche verso i *clients*.

2.7.1. Soluzione *pull*

La soluzione implementata in *Darm-Backend* implica il *pull* da parte del client che utilizza una *url* con segmenti che includono:

- tipo di upload (manuale|remoto);
- jobId;
- step (check_file,load_temp_table,check_data,insert_data,register).

La risposta ha come contenuto un'istanza di jobProgress che è configurato come segue.

```
type JobProgress struct {
    JobId     string `json:"jobId" db:"job_id"`
    Step      string `json:"step" db:"step"`
    Message   string `json:"message" db:"message"`
    Status    string `json:"status" db:"status"`
    Completed bool   `json:"completed" db:"completed"`
}
```

Da sottolineare che il campo *Completed* diventa vero o quando tutti gli steps sono stati chiusi con succcesso o quando la pipeline viene terminata per un errore.

Nell'interfaccia utente del *manual upload* i dati del jobProgress sono utilizzati per evidenziare, con modalità diverse e con colori diversi, il successo o il fallimento dei singoli *steps*, come si può vedere negli screenshots che seguono.

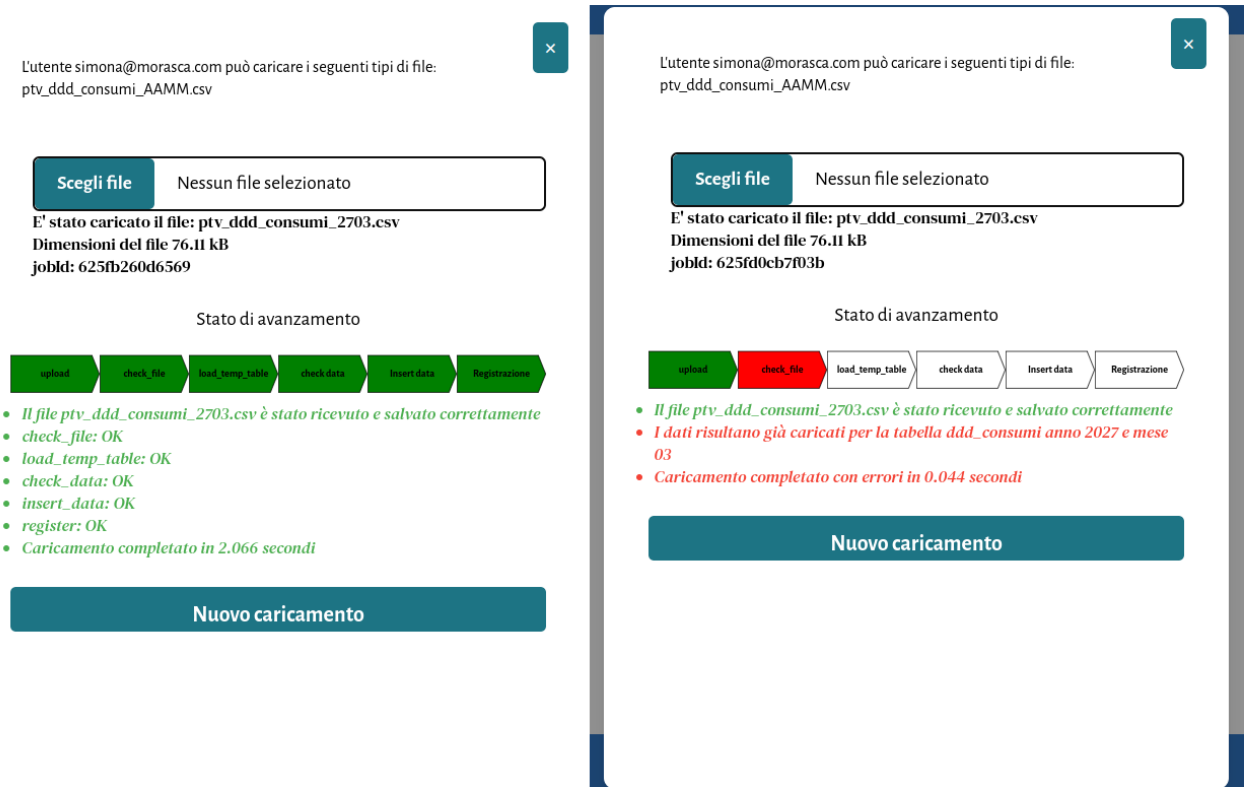

Fig. 10 - Notifiche di successo e di errore

Le due modalità di upload, manuale e remoto, condividono la stessa logica, che è sintetizzata nella figura che segue.

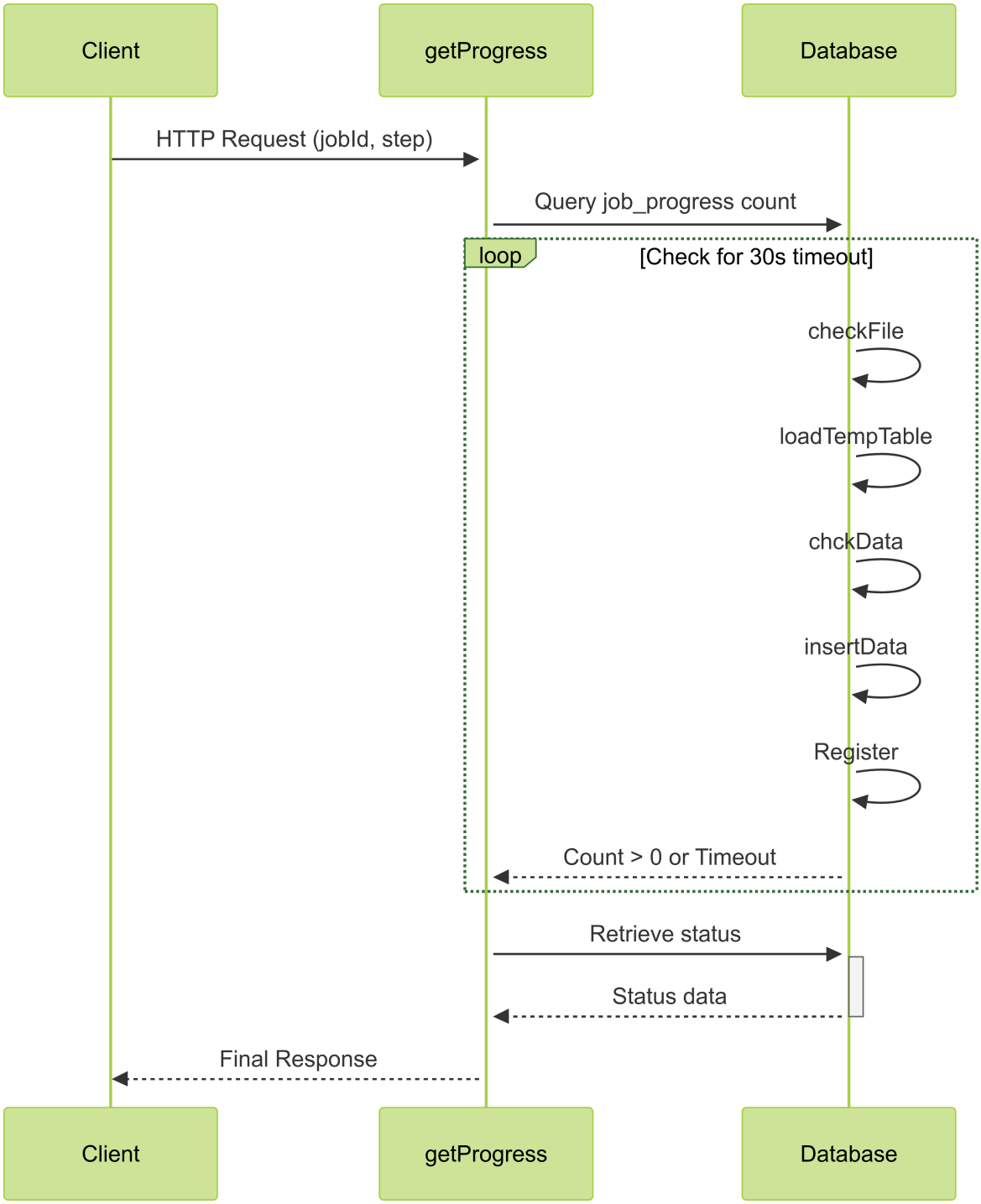

Fig. 11 - Il sistema di notifiche

2.8. Il client per il Remote Upload

Per il *remote upload*, in mancanza di web services già attivi da utilizzare nello sviluppo del relativo package, è stato messo a punto un client che funziona su Linux con database di test ed ha le seguenti caratteristiche:

- si assume che i flussi mensili siano inviati a cura dei providers all'endpoint /api/rest/upload con metodo POST;

- il provider si autentica con un JWT generato dal server;
- la denominazione dei *files* segue la naming convention;
- per la validazione dei nomi dei *files* si utilizza lo stesso metodo seguito per gli utenti con ruolo di *uploader*.

L'implementazione del *remote upload* è stata impostata per una massima integrazione con la gestione della pipeline e delle notifiche. Il flusso evidenziato nel sistema di notifiche è lo stesso degli uploads manuali. Gli endpoints sono diversi ma condividono gli stessi handlers. La presentazione degli *outputs* - almeno nel prototipo sviluppato - è ovviamente diversa e si limita al flusso su STDOUT e all'output del logging come di seguito riportati.

```
INF PostRequest 45  file=ptv_pro_consumi_2703.csv user=obiwankenobi@gmail.com
INF Status 200 OK: Response data: Il file ptv_pro_consumi_2703.csv è stato ricevuto e salvato corrett
onsumi_2703.csv jobId=626b423a943cd user=obiwankenobi@gmail.com
INF Response data: check_file: OK file=ptv_pro_consumi_2703.csv jobId=626b423a943cd user=obiwankenobi

INF Response data: load_temp_table: OK file=ptv_pro_consumi_2703.csv jobId=626b423a943cd user=obiwank

INF Response data: check_data: OK file=ptv_pro_consumi_2703.csv jobId=626b423a943cd user=obiwankenobi

INF Response data: insert_data: OK file=ptv_pro_consumi_2703.csv jobId=626b423a943cd user=obiwankenob

INF Response data: register: OK file=ptv_pro_consumi_2703.csv jobId=626b423a943cd user=obiwankenobi@g

INF pipeline completed file=ptv_pro_consumi_2703.csv jobId=626b423a943cd user=obiwankenobi@gmail.com
```

Fig. 12 - Output del client per il Remote Upload

```
[
    {
        "level": "info",
        "user": "iansolo@gmail.com",
        "file": "rm3_cvc_consumi_2404.csv",
        "time": "2024-10-25T05:41:21+02:00",
        "message": "PostRequest 1 "
    },
    {
        "level": "info",
        "user": "iansolo@gmail.com",
        "jobId": "625459e6b18ac",
        "file": "rm3_cvc_consumi_2404.csv",
        "time": "2024-10-25T05:41:21+02:00",
        "message": "Upload successful: Il file rm3_cvc_consumi_2404.csv è stato ricevuto e salvato correttamente"
    },
    {
        "level": "info",
        "user": "iansolo@gmail.com",
        "jobId": "625459e6b18ac",
        "step": "check_file",
        "status": "success",
        "completed": false,
        "time": "2024-10-25T05:41:21+02:00",
        "message": "check_file: OK"
    },
    {
        "level": "info",
        "user": "iansolo@gmail.com",
        "jobId": "625459e6b18ac",
        "step": "load_temp_table",
        "status": "success",
        "completed": false,
        "time": "2024-10-25T05:41:21+02:00",
        "message": "load_temp_table: OK"
    },
```

```
    {
        "level": "info",
        "user": "iansolo@gmail.com",
        "jobId": "625459e6b18ac",
        "step": "check_data",
        "status": "success",
        "completed": false,
        "time": "2024-10-25T05:41:21+02:00",
        "message": "check_data: OK"
    },
    {
        "level": "info",
        "user": "iansolo@gmail.com",
        "jobId": "625459e6b18ac",
        "step": "insert_data",
        "status": "success",
        "completed": false,
        "time": "2024-10-25T05:41:21+02:00",
        "message": "insert_data: OK"
    },
    {
        "level": "info",
        "user": "iansolo@gmail.com",
        "jobId": "625459e6b18ac",
        "step": "register",
        "status": "success",
        "completed": true,
        "time": "2024-10-25T05:41:21+02:00",
        "message": "register: OK"
    },
    {
        "level": "info",
        "user": "iansolo@gmail.com",
        "jobId": "625459e6b18ac",
        "file": "rm3_cvc_consumi_2404.csv",
        "time": "2024-10-25T05:41:21+02:00",
        "message": "Job processing completed"
    }
]
```


Fig. 13 - Log del client per il Remote Upload

Ai fini della messa in produzione il client dovrebbe essere rilasciato al provider e adattato alle caratteristiche della relativa infrastruttura e nel linguaggio richiesto (php, java, go, rust, etc.)

2.8.1. Implementazioni alternative

La configurazione del client per l'implementazione del *remote upload* può essere oggetto di scelte alternative, quali:

- **il metodo di acquisizione dei dati**. Il client che è stato sviluppato si basa sul POST, quindi la prima alternativa dovrebbe essere un sistema basato sul metodo GET. Tale soluzione potrebbe rientrare in un quadro più generale di pianificazione, che a date prestabilite preleva dall'endpoint del provider il flusso richiesto. Una configurazione;
  - la url ha come parametri i segmenti del filename (azienda-flusso-periodo);
  - server e provider condividono la stessa mappatura delle risorse, secondo i criteri indicati nella *naming convention*, e conseguentemente la stessa mappatura degli *endpoints* e del filesystem;

- **soluzioni basate sul push**.

2.8.2. Il setup del client

Le fasi principali del processo client sono:

1. Upload del file;
2. Attivazione della pipeline;
3. Monitoraggio dell'avanzamento della pipeline.

Alle fasi di cui sopra corrispondono tre endpoints distribuiti in due gruppi di routing, distinti per modalità di upload, come di seguito riportato.

```
r.Group(func(r chi.Router) {
 r.Use(dynamic.Append(app.jwtTAuthMiddleware).Then) // REST API
 r.Post("/api/rest/upload", app.postRestUpload)
 r.Post("/api/rest/start-process/{jobId}", app.postPipelineStarter)
 r.Get("/api/rest/progress/{jobId}/{step}", app.getProgress)
 })

r.Group(func(r chi.Router) {
        r.Use(dynamic.Append(app.authMiddleware,
app.uploaderRoleCheckMiddleware).Then)
        r.Post("/upload", app.postUpload)
        r.Post("/start-process/{jobId}", app.postPipelineStarter)
        r.Get("/progress/{jobId}/{step}", app.getProgress)
    })
```

Da sottolineare il fatto che *manual upload* e *remote upload* condividono gli stessi handlers per le fasi 2. e 3. mentre differiscono, ovviamente, per la fase 1.

In questo modo è stata conseguita la massima integrazione possibile delle due modalità di upload, minimizzando l'impatto sui costi di sviluppo perché le differenze si limitano allo specifico handler della nuova modalità di upload e al middleware che deve essere specifico per sistema di autenticazione.

I providers

L'utente che effettua il *remote upload* è definito *provider* ed ha un profilo rappresentato dalla struttura dati di seguito riportata.

```
type Provider struct {
    Uuid         string   `json:"uuid,omitempty"`
    Username     string   `json:"username"`
    Email        string   `json:"email"`
```

```
    Telefono     string   `json:"telefono"`
    Azienda      string   `json:"azienda"`
    Jwt          string   `json:"jwt"`
    AllowedFiles []string `json:"permissions,omitempty"`
}
```

Nel database la tabella utilizzata è *admin.providers*.

Per l'esecuzione dei *tests* sono stati creati tre providers:

B	C	E	H
username	**email**	**azienda**	**allowed_files**
obiwan	obiwankenobi@gmail.com	ptv	{ptv_sio_sdo_AAMM.zip,ptv_sio_adt_AAAAMMGG.zip,ptv_ddd_consumi_AAMM.csv,ptv_cvc_consumi_AAMM.csv,ptv_pro_c
iansolo	iansolo@gmail.com	rm3	{rm3_ddd_consumi_AAMM.csv,rm3_cvc_consumi_AAMM.csv,rm3_pro_consumi_AAMM.csv,rm3_gel_consumi_AAMM.csv,rr
JangoFett	jangofett@gmail.com	inmi	{inmi_cvc_consumi_AAMM.csv,inmi_gel_consumi_AAMM.csv,inmi_sio_adt_AAAAMMGG.csv,inmi_ddd_consumi_AAMM.csv}

Per la registrazione di un provider viene utilizzata la form di seguito rappresentata e i relativi dati sono salvati nella tabella *admin.providers*.

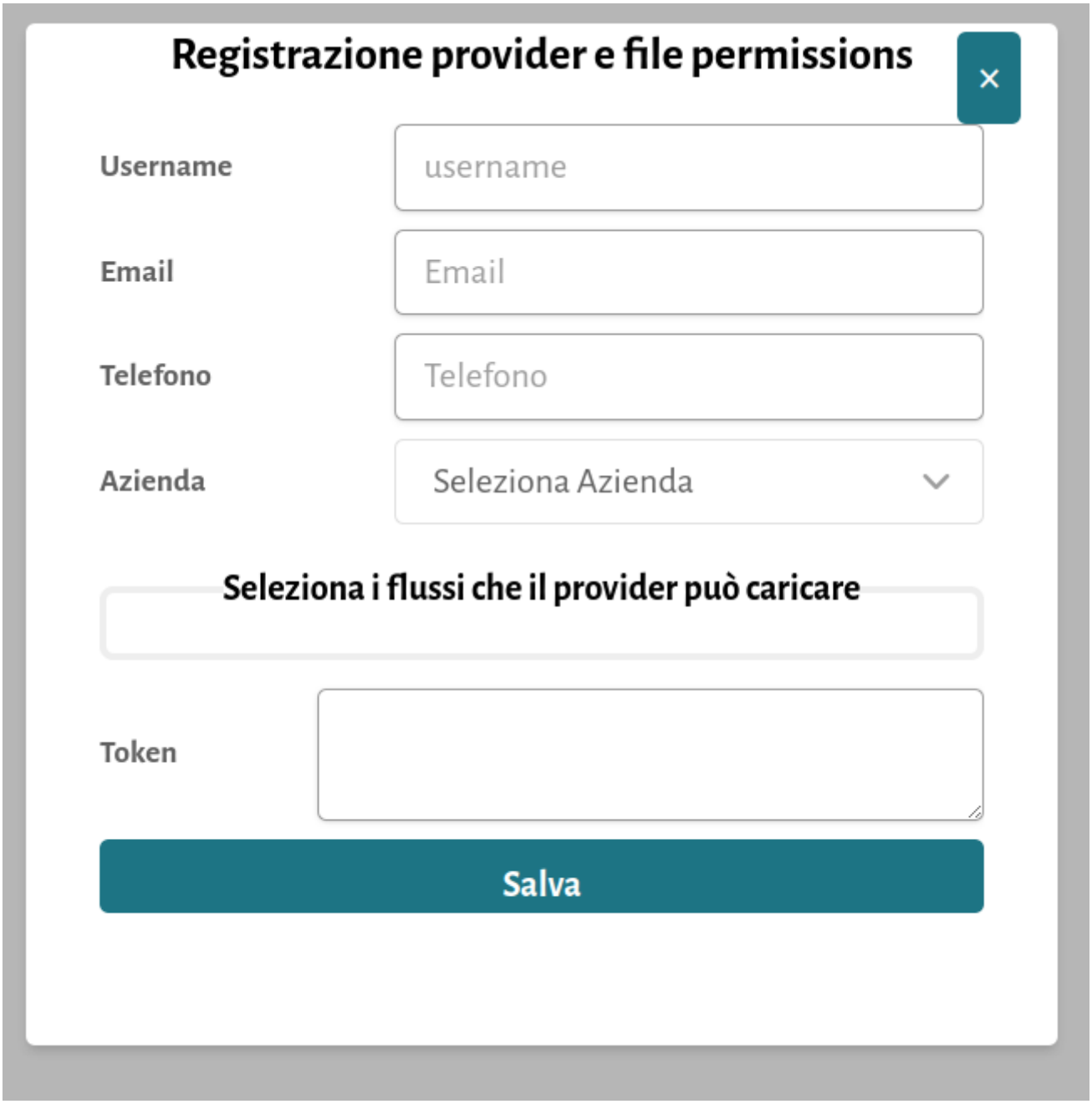

Fig. 14 - Form di registrazione di un provider

Il JWT

Il sistema di autenticazione, come già specificato in un precedente paragrafo, utilizza un JWT generato dal server e verificato ad ogni passaggio dal middleware dedicato. In un ambiente di produzione sono previsti alcuni passaggi che l'implementazione semplificata del prototipo ha omesso (ad es.: autenticazione iniziale attraverso una chiave fornita dal server, rilascio di un JWT temporaneo, procedure di refresh, etc). L'implementazione ha utilizzato i seguenti *claims*

```
type Claims struct {
    Username string `json:"username"`
    Email    string `json:"email"`
```

```
    jwt.RegisteredClaims{
            Issuer:    string `json:"iss,omitempty"`,
            ExpiresAt: *NumericDate `json:"exp,omitempty"`
            IssuedAt:  *NumericDate `json:"iat,omitempty"`
        }
}
```

Un middleware dedicato controlla gli *endpoints* e i successivi passaggi, verificando sul database il JWT che identifica il provider.

I dati

Per il test è stato predisposto un elenco di 187 files, prevalentemente quelli già caricati per i vari flussi dagli utenti delle tre aziende. L'elenco è implementato come file json strutturato come di seguito rappresentato.

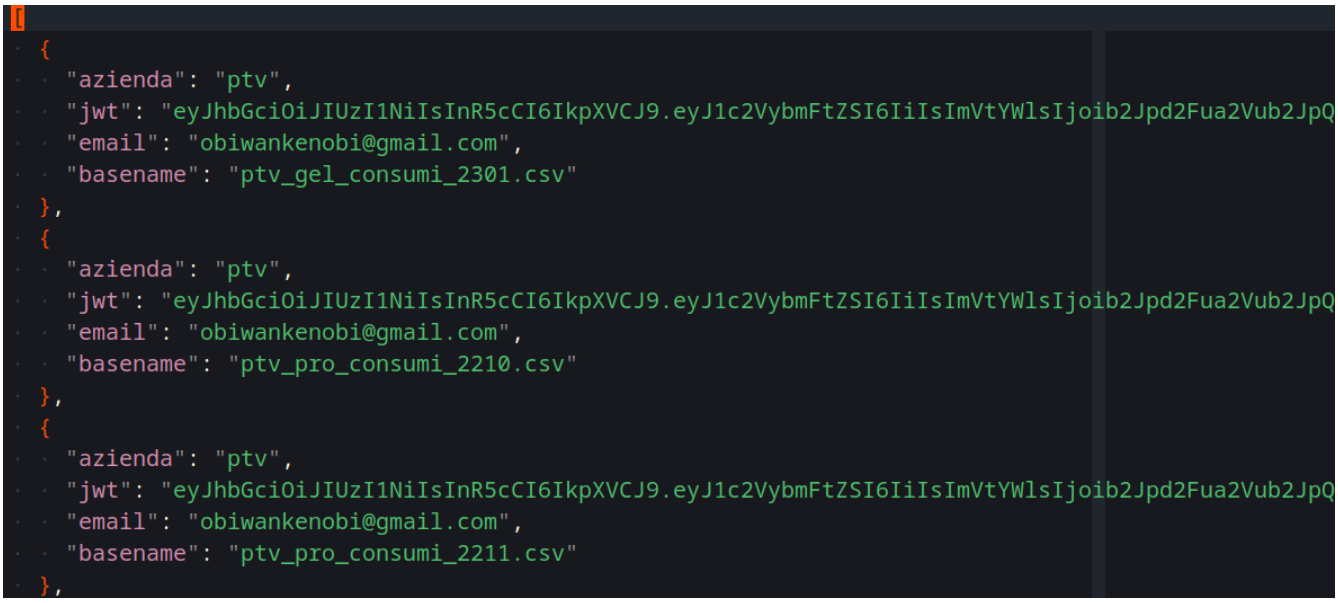

Fig. 15 - Elenco dei files per il test del client

I *tests* eseguiti e i relativi risultati

Come sopra specificato, il client può essere utilizzato per una richiesta specifica oppure per lanciare un numero definito di richieste. Il primo lotto di *tests* eseguiti sono stati prevalentemente orientati al caricamento dell'intero batch dei 187 files, operazione che impiega una decina di minuti e un output su file json. Con il supporto di *jq* è stata messa a punto una rudimentale analisi dei dati per quantificare gli errori per tipologia e le interruzioni del caricamento dovute *timeout* che si attestano su un range dal 4% al 13%, più frequenti per i *files* di maggiori dimensioni (es. 50.000 righe) ma anche con numerosi casi di dimensioni molto più contenute. Dall'esame dei log o dall'osservazione diretta e contemporanea della console del client e della console del server, risalta il ritardo nell'avanzamento dei singoli steps tra i due processi, dove ad esempio nel log del server la pipeline risulta completata, mentre il client è ancora in attesa del risultato di `check_data` o di `register`.

2.8.3. Le metriche

Tenendo conto di quanto sopra riportato sul ritardo della notifica rispetto al processo principale, è stata creata la tabella *sse_metrics* per tracciare i tempi di latenza. La tabella viene aggiornata contestualmente all'invio della notifica da parte del database.

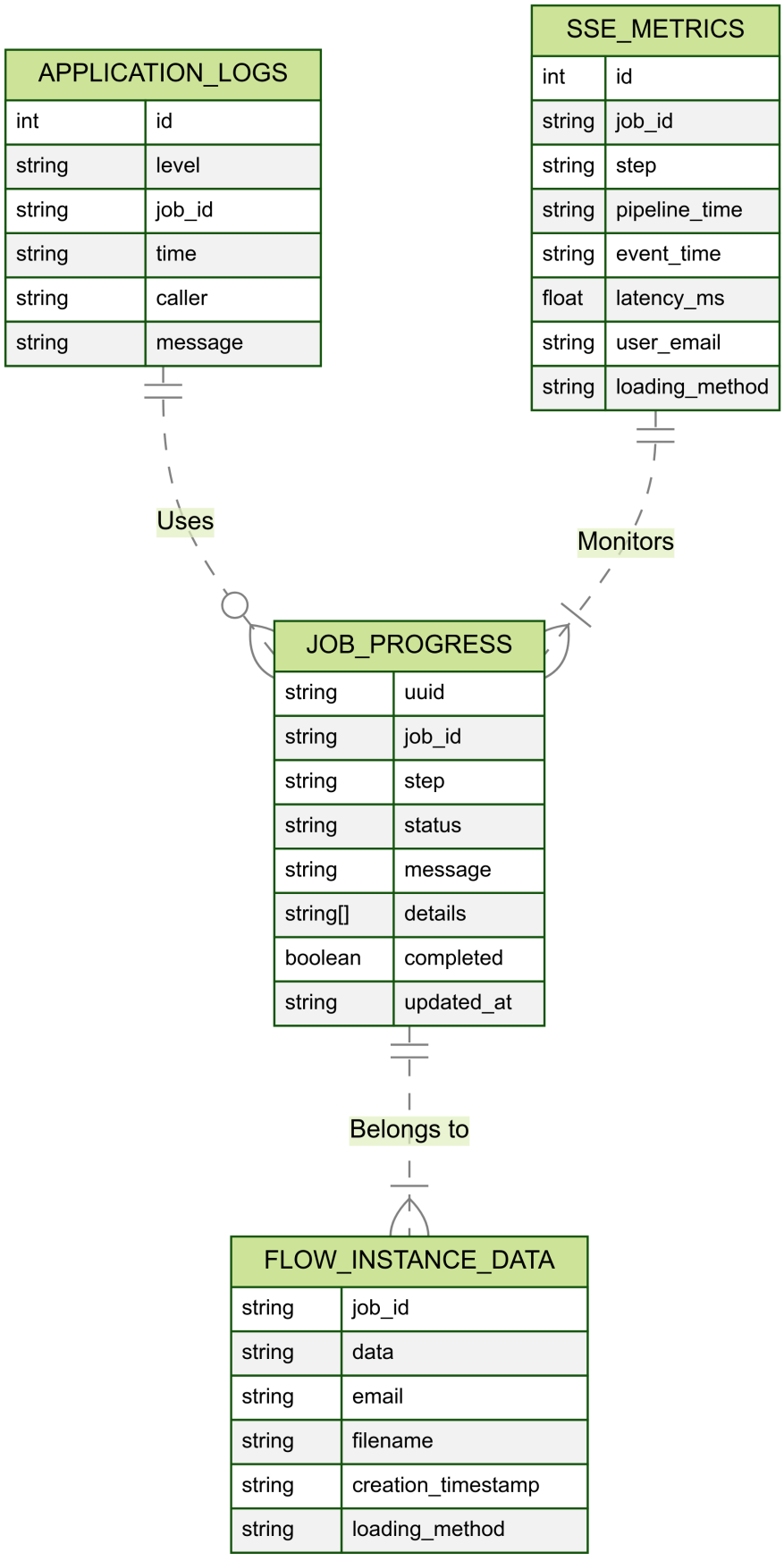

Fig. 16 - La tabella delle metriche

2.9. La gestione degli errori

Gli errori sono stati classificati in due tipologie:

- *errori di dominio*: sono errori rilevati nei dati che possono riguardare un formato del file non corretto oppure la violazione di *check constraints* rilevata in fase di caricamento nella tabella finale (es. la data di trasferimento del paziente che precede la data di ricovero); questi errori sono notificati all'utente che ha caricato il file e comportano l'annullamento del caricamento;
- *errori di sistema*: sono errori rilevati durante l'esecuzione del caricamento, che possono essere dovuti a problemi di connessione al database, a problemi di accesso al file, a problemi di connessione al server di archiviazione, etc.

In entrambi i casi l'errore viene registrato nel database unitamente ai dati di contesto (tipo di errore, job_id, timestamp, funzione e numero di riga, etc.).

2.10. Logging e tracciabilità

Il sistema di log utilizzato è *zerolog* che presenta le seguenti funzionalità:

- è strutturato per livelli di log (debug, info, warning, error, fatal);
- in fase di avvio dell'applicazione è possibile definire un livello minimo di log;
- possono essere impostati diversi tipi di output (console, file e database).

L'implementazione è stata realizzata con due tipi di output: console e database. La tabella in cui vengono registrati i log è *admin.application_logs*.

2.11. L'interfaccia web

2.11.1. Home page

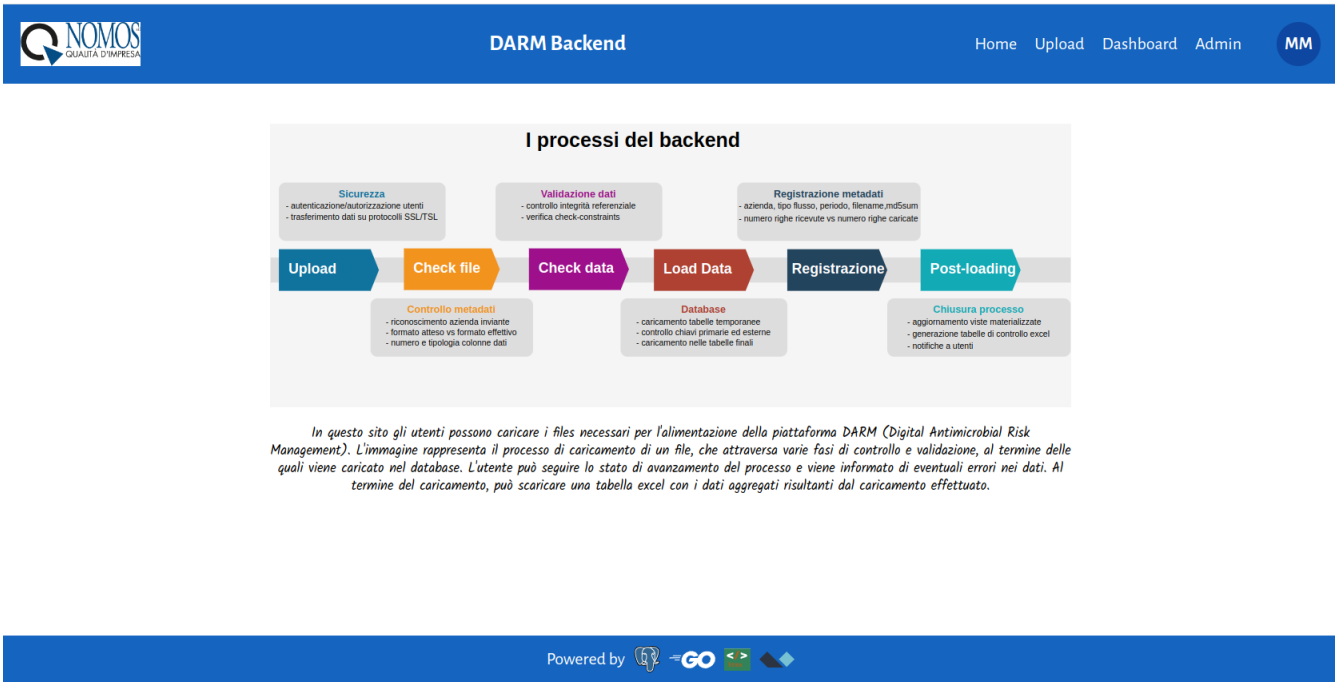

Fig. 17 - La home page

La home page è pubblica

La navbar

La *navbar* si presenta con opzioni diverse in relazione al ruolo dell'utente. L'utente con ruolo di *uploader* vede *Upload* e *Dashboard* , mentre l'utente con ruolo *admin* vede anche il menù *Admin*

2.11.2. Upload

Solo gli utenti registrati e autorizzati al caricamento dei *files* hanno accesso all'*Upload*. All'utente anonimo che tenta di entrare nell' *Upload* viene richiesto di effettuare il login.

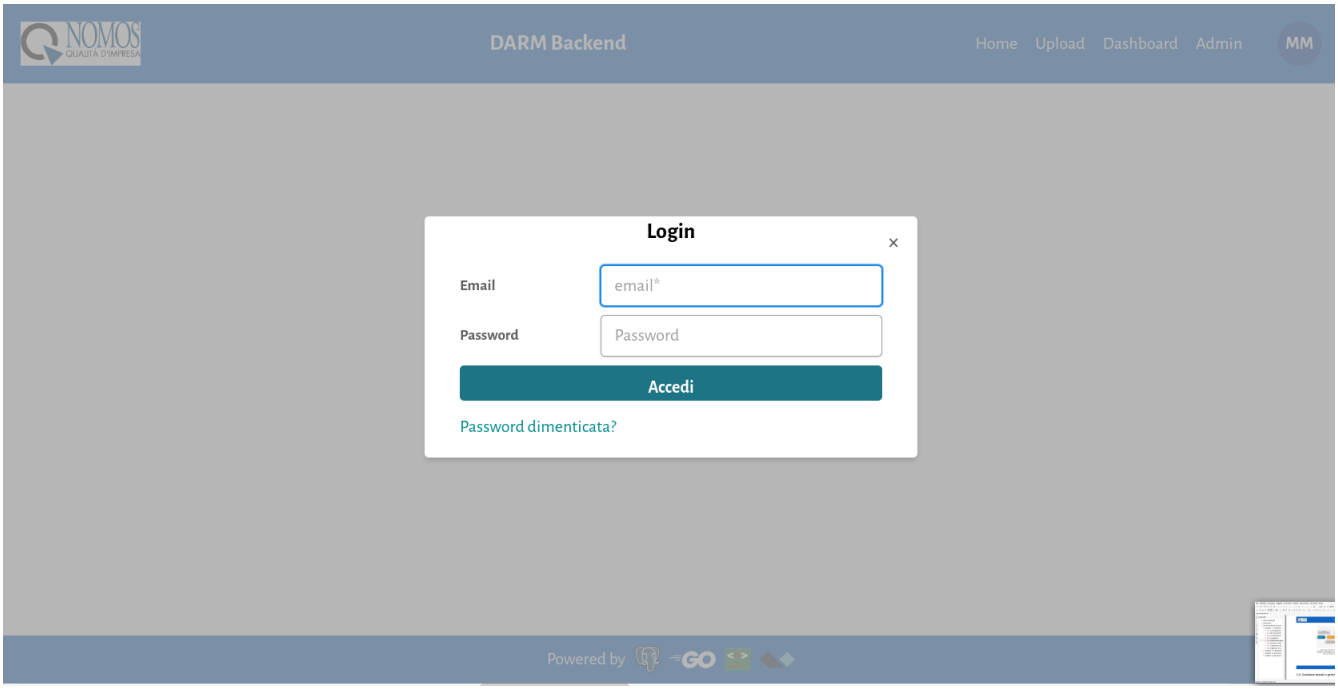

Fig. 18 - La form di login

Superato il login, se si verifica che l'utente è autorizzato all'upload, si presenta l'apposita form come rappresentata nell'immagine che segue:

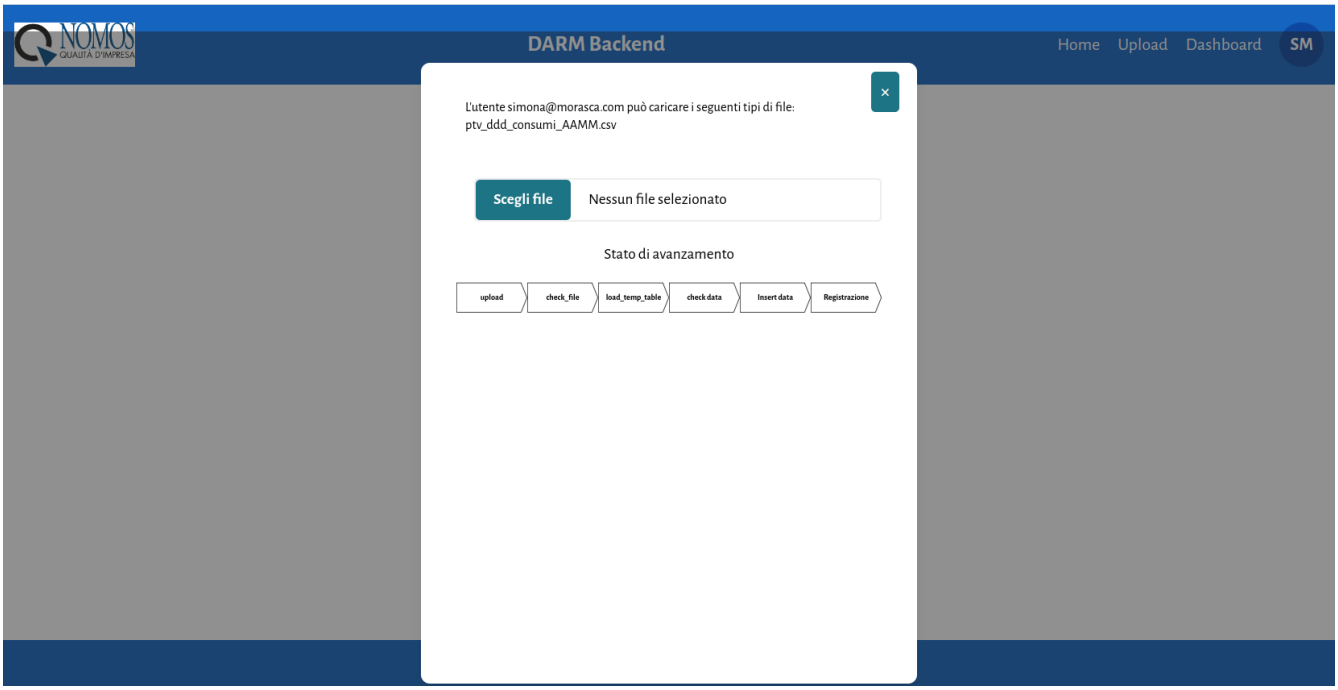

Fig. 19 - La form di upload

L'informazione contenuta nella form vuota è l'indicazione del tipo di *files* che l'utente può caricare. Se l'utente seleziona dal filesystem un file dal nome non conforme, il processo viene interrotto con un messaggio di errore e il file non viene caricato.

Se viene selezionato un file conforme alle autorizzazioni utente, in caso di caricamento andato a

buon fine, la form di *Upload* si presenta come nella figura seguente.

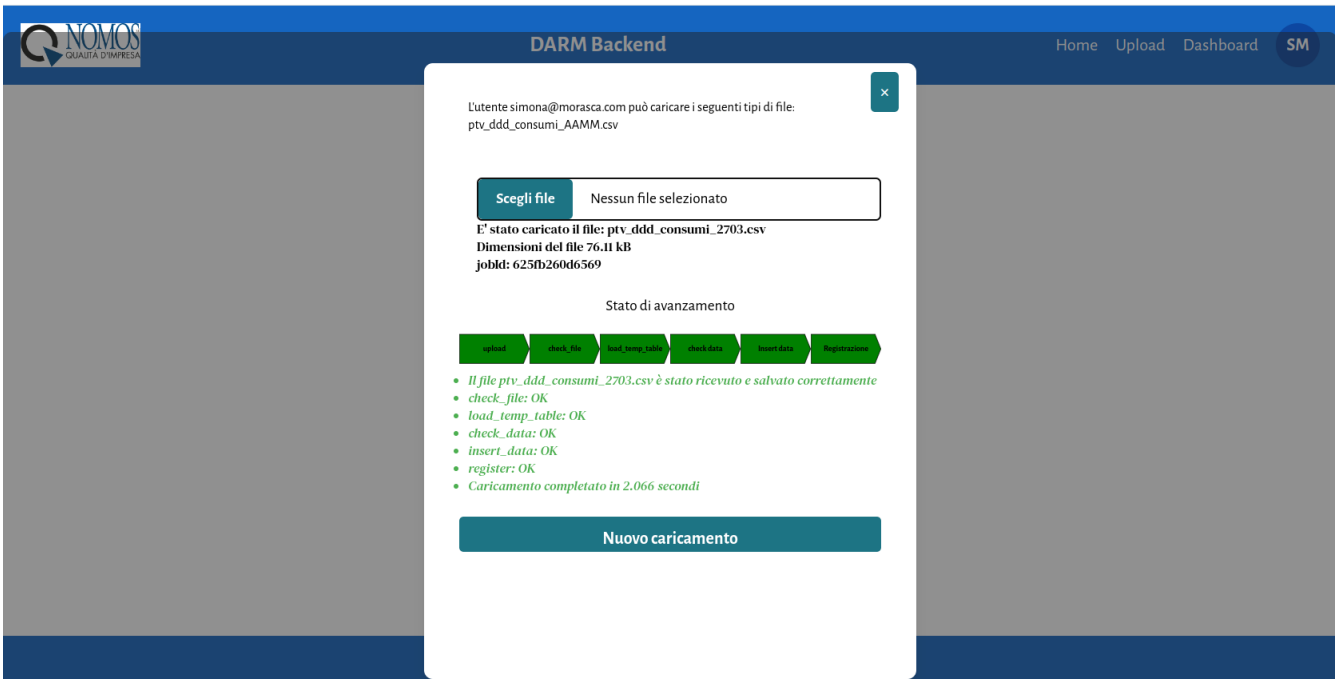

Fig. 20 - Upload completato con successo

In caso di errori, l'errore viene notificato all'utente e la pipeline viene interrotta.

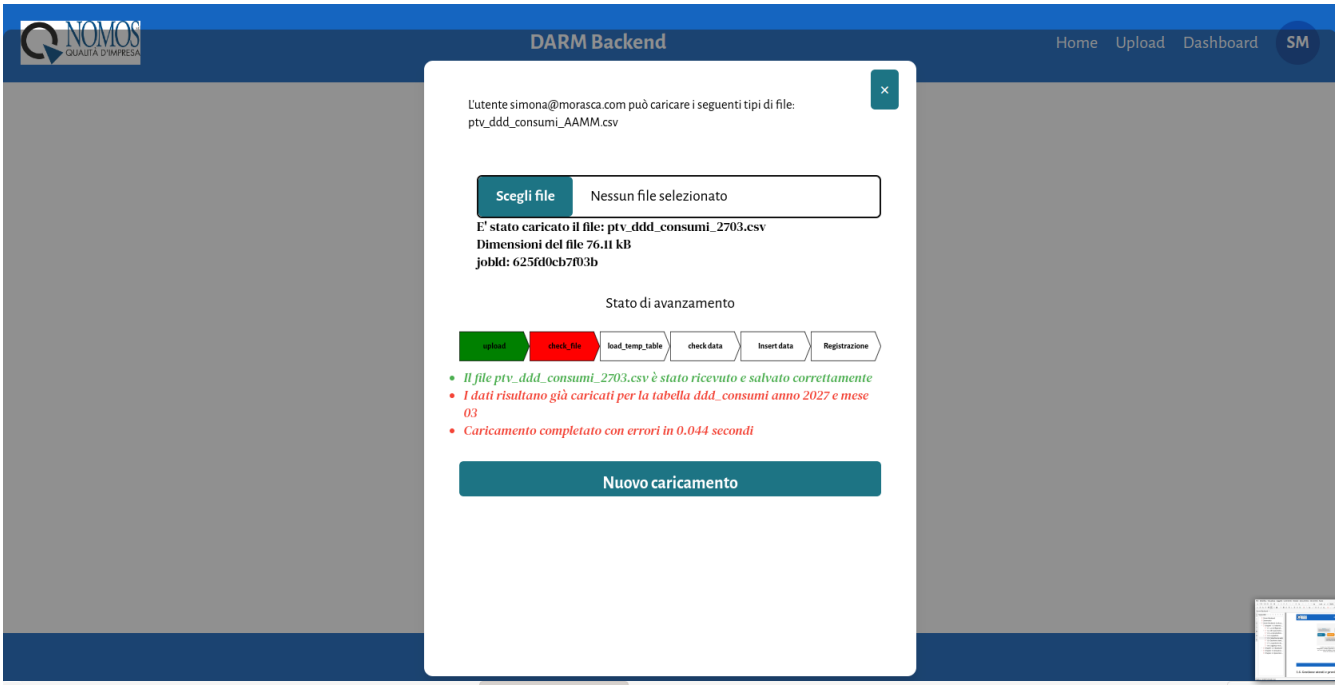

Fig. 21 - Upload non completato per errori

Ovviamente gli esempi di cui sopra sono stati effettuati in un ambiente di test con utenti fittizi e file dati appositamente predisposti per non interferire con il database di produzione.

2.11.3. Dashboard

La pagina della dashboard si presenta con opzioni diverse a seconda del ruolo dell'utente. L'utente *uploader* accede al log dei caricamenti e ai *files* che può scaricare, come evidenziato nella figura che segue.

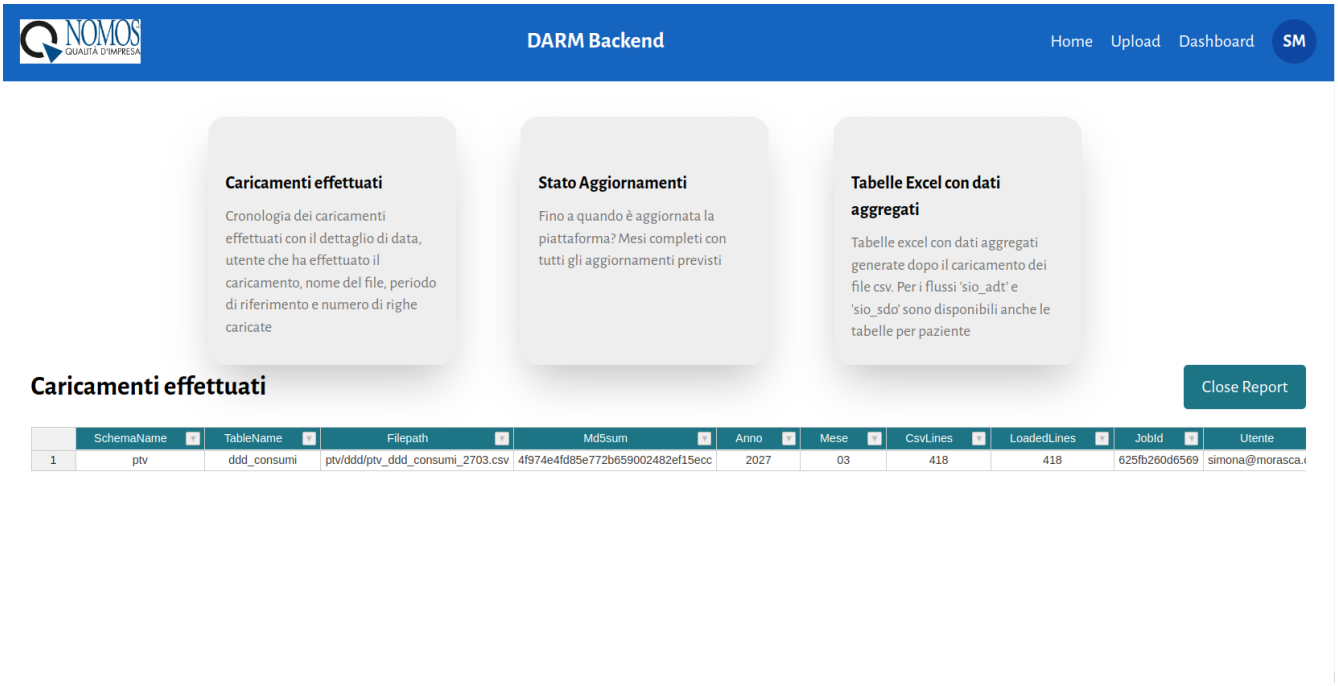

Fig. 22 - La dashboard dell'utente uploader

L'utente *admin* può accedere a una molteplicità di viste sui dati

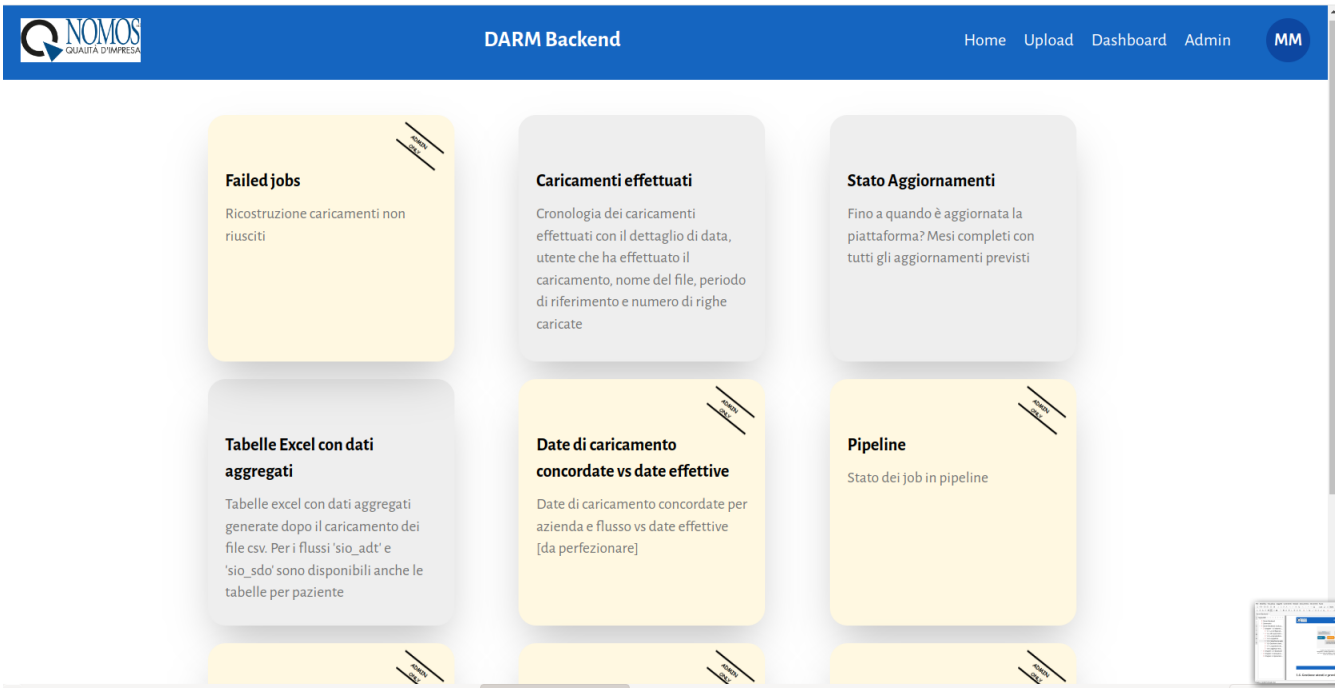

Fig. 23 - La dashboard dell'utente admin

2.11.4. Admin

L'utente *admin* può accedere al menù *Admin* che consente di:

- creare un nuovo utente;
- creare un nuovo provider;
- creare un nuovo report;

Creazione di un nuovo utente

La creazione di un nuovo utente è disponibile solo per un utente *admin* perché consente di assegnare le autorizzazioni necessarie per accedere alle funzionalità dell'applicazione. Ci sono due livelli di autorizzazione:

- il ruolo, che può essere *uploader*, *observer* e *admin* come rappresentato in fig. 24;

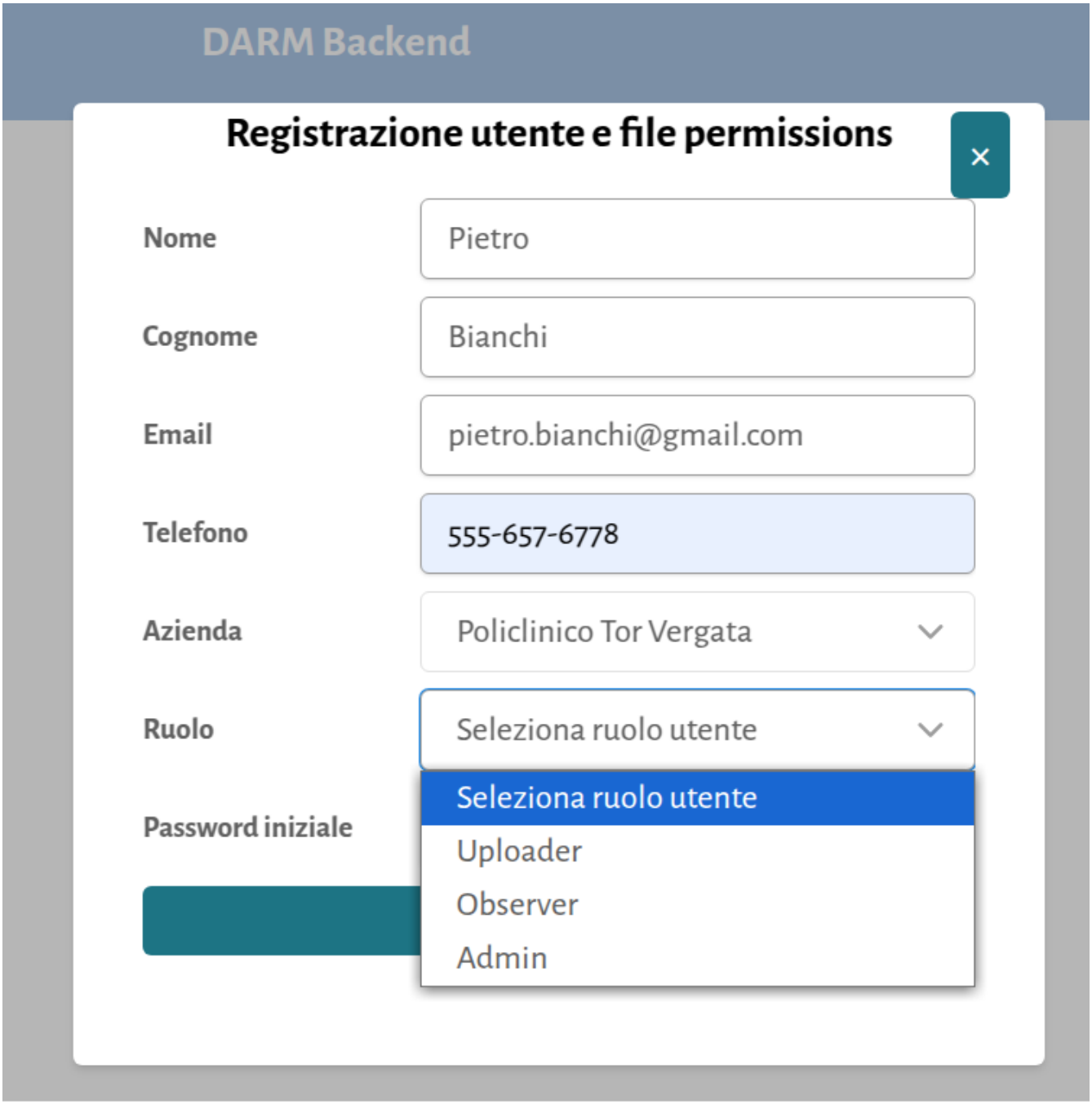

Fig. 24 - Creazione di un nuovo utente e asegnazione del relativo ruolo

- se il ruolo è *uploader*, sono definiti i flussi che l'utente può caricare, come rappresentato in fig. 25.;

Fig. 25 - Permessi relativi ai flussi che il nuovo utente può caricare

Infine viene assegnata una password iniziale che potrà essere cambiata dall'utente al primo accesso attraverso la *form* di modifica password.

Creazione di un nuovo provider

Nel prototipo sviluppato per il *remote upload* la *form* di creazione di un nuovo provider è molto simile a quella di creazione di un nuovo utente *uploader*, soprattutto per quanto riguarda l'assegnazione delle *file permissions*.

Il report manager

Per rispondere all'esigenza di reportistica sugli *operation data* del backend, è stato sviluppato un generatore di codice che acquisisce alcuni parametri da un'apposita form e genera automaticamente i metadati per la creazione del report *runtime*, con un approccio *low code* che consente all'utente di creare report senza conoscenze specifiche di programmazione ma in grado di definire le query necessarie per estrarre i dati.

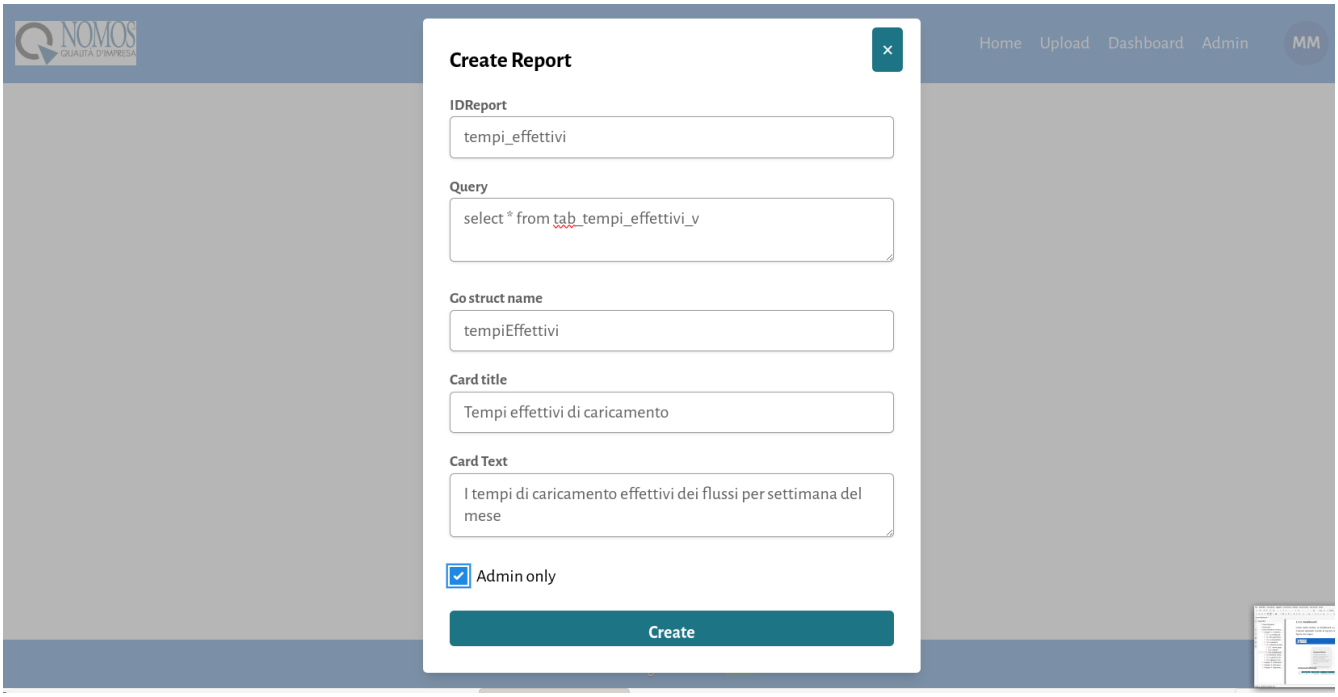

Fig. 26 - Creazione di un report

Una volta confermata la creazione del report, vengono generate le seguenti componenti in base ai parametri forniti dall'utente:

- le strutture dati e le interfacce in Go per lo specifico report;
- un file json con le specifiche per la creazione runtime della tabella di visualizzazione dei dati;
- la *card* html per la visualizzazione nella dashboard del titolo e della descrizione del report; se il report è destinato all'utente *admin*, la card riporta la scritta **admin only** e un background di colore diverso rispetto ai report disponibili all'utente *uploader*. Una volta completata la creazione del report, la relativa *card* appare nella dashboard e l'utente può visualizzare i dati come nel *screenshot* che segue.

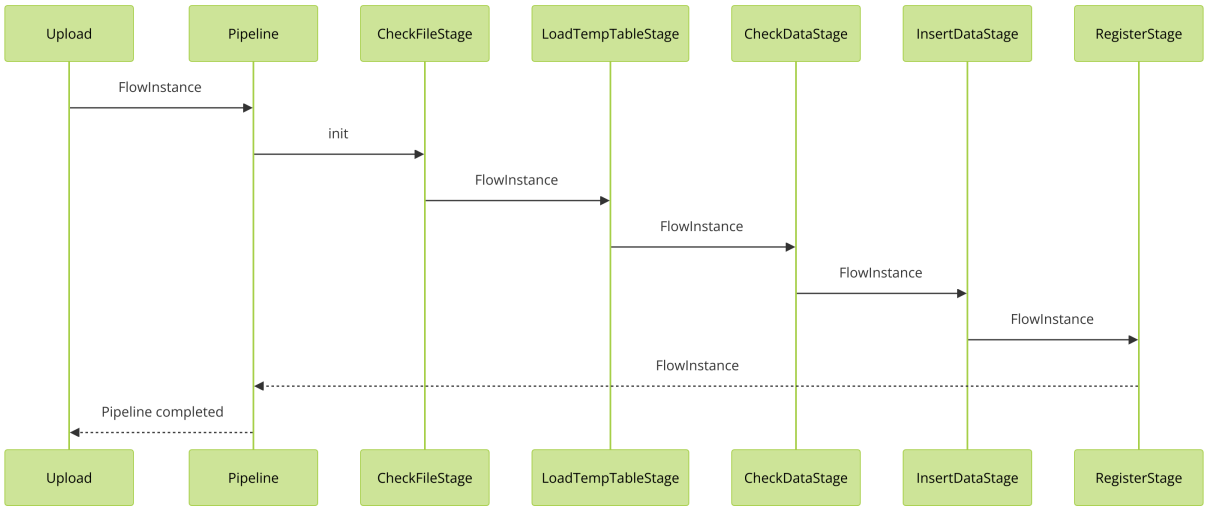

Fig. 27 - Report Tempi Effettivi

L'implementazione del report generator è ancora in una forma basilare per quanto riguarda l'interfaccia utente: ad esempio non esistono *forms* per la modifica e la cancellazione del report e la presentazione dei dati richiederebbe alcuni miglioramenti (es. ricerche complesse). Tuttavia, anche in questa fase iniziale il *report generator* è apprezzabile dagli utenti per la sua semplicità e per la rapidità con cui è possibile creare un report e visualizzare i dati.

Chapter 3. Il database

Il database è implementato con Postgresql v.17.2 ed è organizzato con un approccio *multitenant*

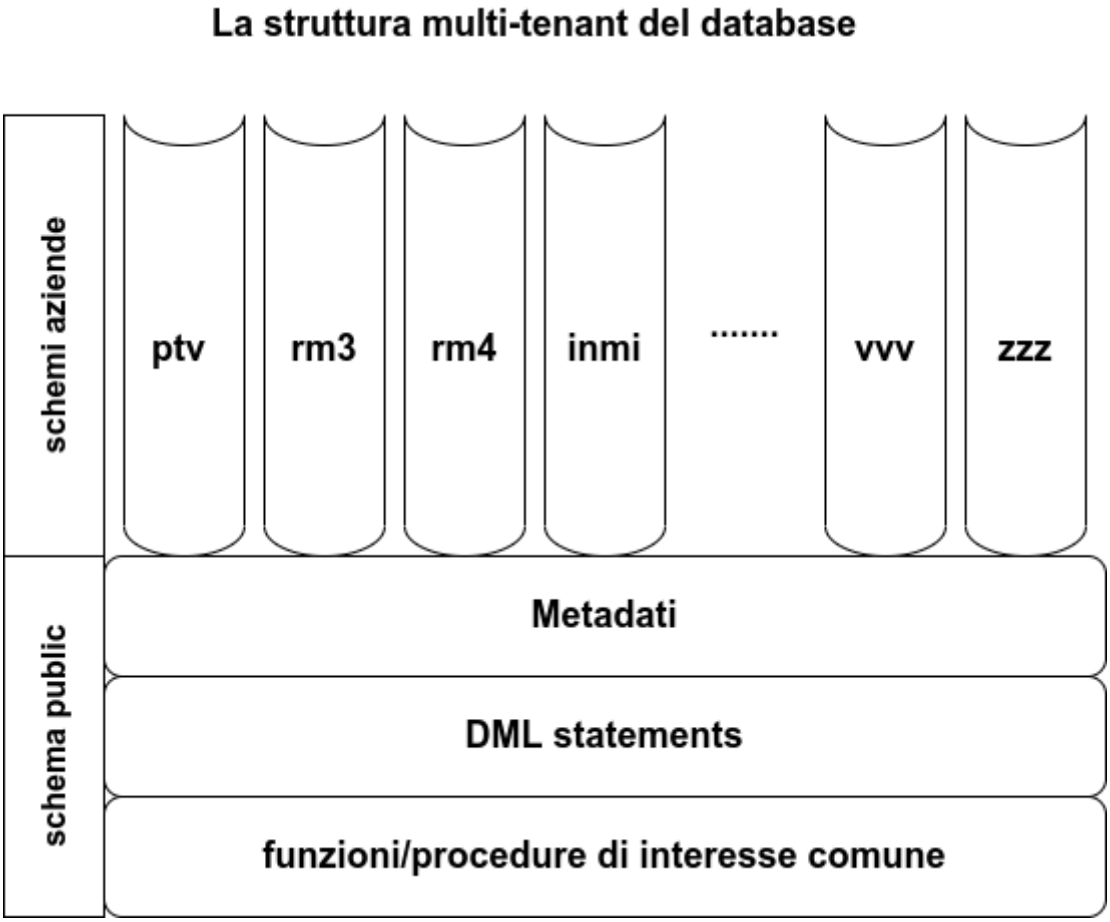

Fig. 30 - Il database multitenant

L'organizzazione del database è basata sugli schemi del database ICA. Lo **schema public** contiene:

- metadati relativi alle aziende, alle istruzioni di aggiornamento dei dati e al log di caricamento dei singoli flussi;
- funzioni/procedure di interesse comune;
- viste di interesse comune che possono riguardare lo stato dei caricamenti o l'individuazione nelle tabelle temporanee delle violazioni ai *constraints*.

3.1. Lo schema azienda

3.1.1. Il modello dei dati

Un'azienda tipo può essere rappresentata come nello schema che segue, dove si evidenziano sia le tabelle target sia le tabelle di referenza con le quali le tabelle target sono collegate tramite chiavi esterne.

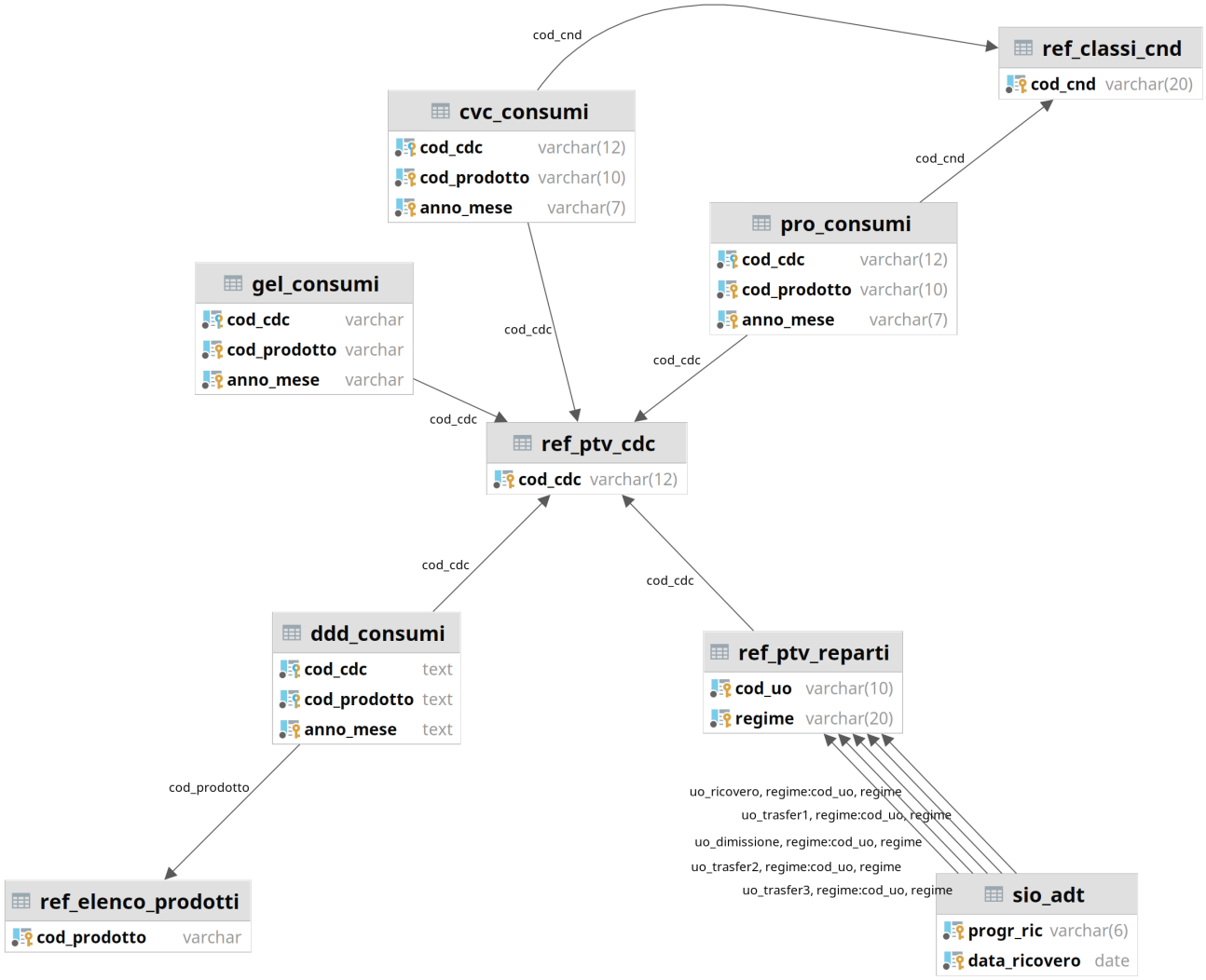

Fig. 31 - Lo schema azienda

3.1.2. Le tabelle temporanee

Ad ogni tabella target è associata una tabella temporanea denominata *temp_[tabella_target]* che viene utilizzata per:

- il caricamento iniziale dei dati nella fase di LoadTempTable della *pipeline*, con un tracciato che differisce dalla relativa tabella target perché tutti i campi di tipo *numeric* o *date* sono di tipo testo perché dovranno essere trasformati nel formato atteso dal database. Ad esempio, nei dati del file csv si può trovare un valore quale **1.634,5** che per essere accettato nel campo numerico di destinazione deve essere trasformato in **1634.5**, mentre la data **10/05/2023** o **10-mag-2023** deve essere trasformata in **2023-05-10**;
- consentire un controllo preliminare delle chiavi esterne ed inserire eventuali chiavi mancanti, in esecuzione della fase CheckData della *pipeline*.

3.2. Lo schema *public*

3.2.1. I metadati

Le tabelle di metadati sono *dd_tables_ e* dd_csv_files_, entrambe rappresentate nel diagramma ER di seguito riportato che evidenzia sia il tracciato che la relazione tra le due tabelle.

_dd_tables			
varchar	schema_name	PK	Primary key part
varchar	table_name	PK	Primary key part
varchar(10000)	temp_select		
varchar(30)	vista		
varchar(200)	descrizione		
text	insert_sql		
text	temp_table_ddl		
varchar	fkeys_check		
text	insert		
varchar	flusso		
int	monthly_deadline		

is referenced by

_dd_csv_files			
int	id	PK	
varchar(30)	schema_name	FK	Foreign key to _dd_tables
varchar(30)	table_name	FK	Foreign key to _dd_tables
varchar(200)	filepath		
varchar(100)	md5sum		
varchar(4)	anno		
varchar	mese		
date	last_update		Defaults to current date
varchar	status		
varchar	file_errors		
int	csv_lines		Defaults to -1
int	loaded_lines		Defaults to -1
varchar	job_id		
varchar(40)	utente		
text	loading_method		

Fig. 32 - Le tabelle dd_tables e dd_csv_files

I campi di maggiore interesse di *_dd_tables* sono:

- *schema_name*: nome del schema che identifica l'azienda;
- *table_name*: nome della tabella di destinazione;
- *flusso*: il nome del flusso dati;
- *insert_sql*: contiene l'istruzione SQL per l'inserimento dei dati nella tabella di destinazione che viene utilizzata nella fase *insert_data* della pipeline;
- *fkeys_check*: contiene l'istruzione SQL per il controllo delle chiavi esterne che viene utilizzata nella fase *check_data* della pipeline.

I campi di maggiore interesse di *_dd_csv_files* sono:

- *schema_name* e *table_name* : nome dello schema e della tabella in cui sono stati caricati i dati;
- *filepath*: il percorso completo del file caricato;
- *md5sum*: lo hash comunemente usato per l'identificazione univoca dei files;
- *anno* e *mese*: anno e mese di competenza dei dati;
- *csv_lines* : numero di righe del file csv esclusa la riga di intestazione;
- *loaded_lines*: numero di righe del file caricato;
- *status*: lo stato del file (unloadable,loadable,registered,loaded);
- *job_id*: identificativo del job di caricamento;
- *user*: utente che ha effettuato l'upload;
- *loading_method*: metodo di caricamento del file (manuale o remoto);
- *last_update*: data e ora dell'operazione.

3.2.2. Le funzioni a supporto del caricamento dei dati

Funzioni di trasformazione

Sono state implementate le seguenti funzioni di trasformazione:

- *anno_mese(text)*: restituisce la stringa presente nel file csv (es. mag-2024) nel formato standard AAAA-MM(es. 2024-05);
- *anno_mese_dim(date)*: accetta come parametro una data e restituisce il corrispondente periodo in formato standard AAAA-MM(es. 2024-05), funzione utilizzata nel flusso SIO per attribuire il periodo di competenza dei dati in base alla data di dimissione del paziente;
- *text2num(text)*: restituisce il valore numerico di una stringa di testo, eliminando eventuali punti e virgole e sostituendo la virgola con il punto (es. 11.432,65 → 11432.65).

Funzioni di controllo e di inserimento dati

A differenza delle funzioni di trasformazione che agiscono solo su singoli campi, altre funzioni agiscono sull'intero dataset:

- *check_fks(varchar,varchar,varchar)*: questa funzione, che supporta la fase CheckData della pipeline, accetta come parametri:
  - il nome dello schema;
  - il nome della tabella di destinazione;
  - il jobId che traccia il processo di caricamento.

Il controllo delle *foreign keys* viene effettuato tramite vista dedicata associata alla tabella di destinazione, come rappresentato nella figura che segue.

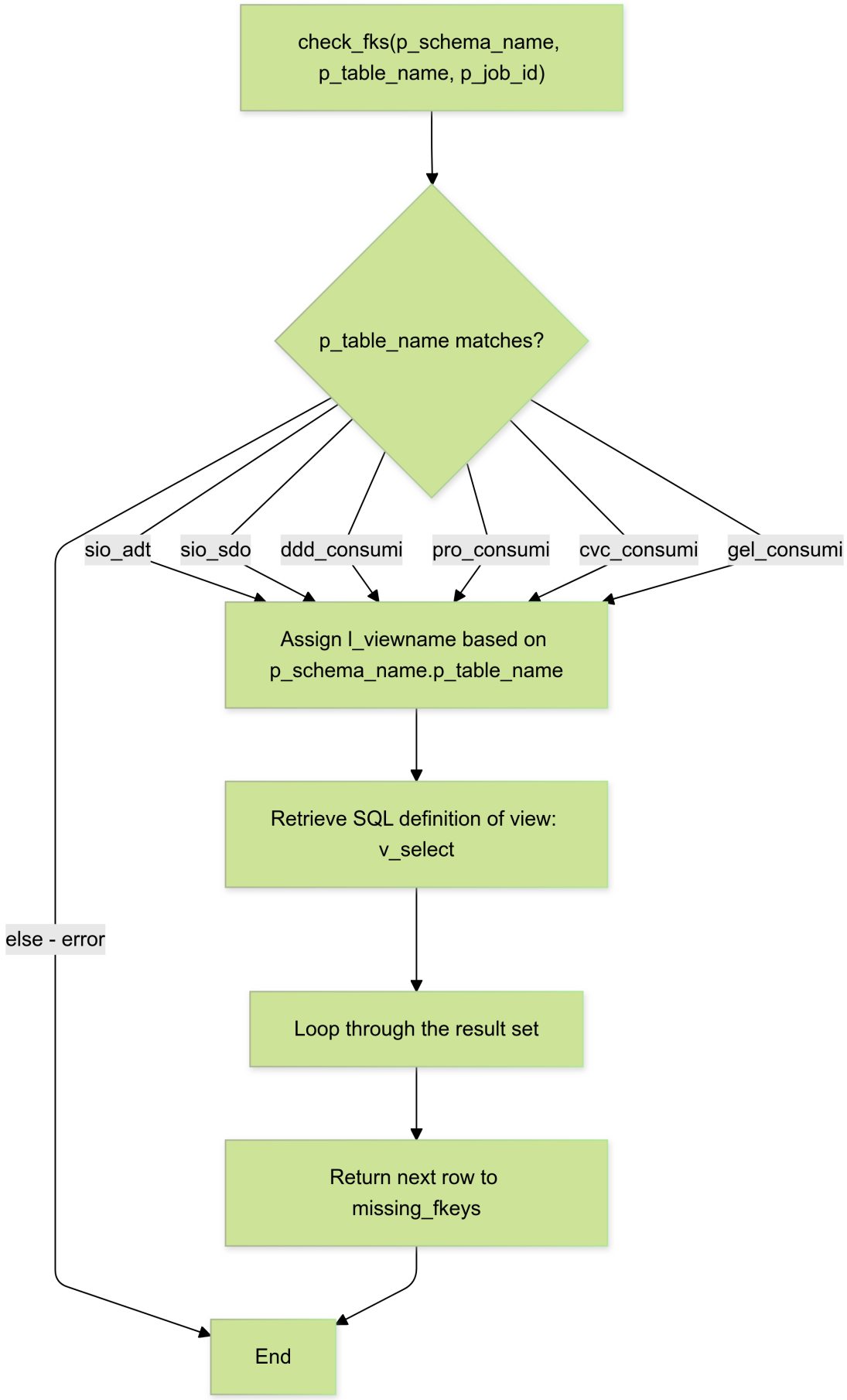

Fig. 33 - Controllo delle chiavi esterne

La funzione restituisce un *recordset* di tipo *missing_fkeys* che alimenta la tabella *_dd_missing_fkeys* in cui sono riportati i tipi di chiavi mancanti (es. centro di costo, unità operativa, codice prodotto, etc), il valore mancante, il jobId e la data di inserimento.

- *insert_data_from_temp_table(varchar,varchar,varchar)*: questa funzione, che supporta la fase InsertData della pipeline, accetta come parametri:
  - il nome dello schema;
  - il nome della tabella di destinazione.

Restituisce una stringa con due valori separati da un delimitatore '|' e i seguenti possibili risultati:

- se ci sono stati errori il primo valore è -1 mentre il secondo valore riporta il codice di errore (es. -1| 23505);
- se l'inserimento è andato a buon fine il primo valore riporta il numero di *records* inseriti, mentre il secondo valore è *null*.

3.3. Schema admin

Lo schema *admin* contiene le tabelle di metadati che supportano la gestione dell' applicazione:

- users, che oltre ai dati identificativi contiene un array con i tipi dei file che l'utente può caricare, se gli è stato assegnato il ruolo di uploader;
- providers, che oltre ai dati di contatto contiene il token jwt per l'autenticazione e un array con i tipi dei file che il provider può caricare da remoto;
- application_logs;
- request_logs;
- sessions;
- job_progress;
- flow_instance_data;

3.4. Prepared statements

Tutte le query utilizzate per la gestione della *pipeline* sono riportate nel file *statements.yml* che viene letto in fase di avvio dell'applicazione e caricato in memoria come hashmap, in cui ogni query è identificata da un nome univoco, completa dei *placeholders* da sostituire con gli opportuni parametri in fase di esecuzione. Questa impostazione è utile sotto vari profili:

- facilita la manutenzione delle query, in quanto è sufficiente modificare il file *statements.yml* e non è necessario modificare il codice Go;
- permette di ottimizzare l'esecuzione delle query attraverso le funzionalità dei *prepared statements* che consentono di compilare le query una sola volta e riutilizzarle più volte con diversi parametri,

- separa nettamente la gestione dei dati dalla logica applicativa a tal punto che **l'intero processo di caricamento potrebbe essere gestito manualmente eseguendo in modo sequenziale le query contenute nel file *statements.yml*** o, se necessario, potrebbe facilitare il trasferimento a una diversa piattaforma applicativa della gestione dei dati.

3.5. Backup periodico

In base a un *cronjob* settimanale vengono eseguiti periodicamente due tipi di backup del database *ica*: *ica-schema-{data}.dmp* che contiene solo i metadati e *ica-{data}.dmp* che contiene dati e metadati.

3.6. Setup database di test

In molti casi disporre di un database di test è fondamentale per testare nuove funzionalità che si intende introdurre nell'applicazione Go o nel database. I requisiti per il setup del database di test sono:

- l'accesso come superuser postgres;
- la disponibilità dei due backup più recenti del database di produzione, *ica-schema-{data}.dmp* e *ica-{data}.dmp*.

Gli *steps* per il setup del database di test sono:

1. postgres@NOMOS-Computi:~$ psql -d ica;
2. postgres@ica=DROP DATABASE ica_test;
3. postgres@ica=CREATE DATABASE ica_test; \q
4. postgres@NOMOS-Computi:~$ psql -d ica_test < /home/nomos/backup/ica-schema-24-10-2024.dmp;
5. postgres@NOMOS-Computi:~$ psql -d ica_test < /home/nomos/backup/ica-24-10-2024.dmp;
6. postgres@NOMOS-Computi:~$ psql -d ica_test < ./truncate_tables.sql.

L'intero procedimento richiede pochi minuti [1] e rende disponibile un database in cui gli schemi *public* e *admin* sono già popolati con i metadati necessari, mentre gli schemi aziendali contengono solo tabelle dati vuote e tabelle di referenza già popolate. Viene troncata anche la tabella *"_dd_csv_files"* su cui si registrano i caricamenti andati a buon fine per evitare che i *tests* di caricamento vengano terminati con il seguente errore:

> i dati risultano già caricati per la tabella ... e per il periodo ...

Lo script *truncate_tables.sql* contiene le istruzioni per cancellare i dati delle tabelle temporanee e delle tabelle target degli schemi aziendali esistenti. L'aggiornamento dello script in occasione dell'attivazione di una nuova azienda può essere eseguito manualmente, aggiornando l'array degli schemi (`schemas TEXT[] := ARRAY['ptv', 'rm3', 'inmi','rm1'];`) oppure attraverso un apposito

script bash passando come argomento l'elenco degli schemi aziendali:

Creazione dinamica del file truncate_tables.sql

```
/var/lib/postgresql/truncate_tables.sh ptv rm3 inmi rm1
```

A questo punto è possibile completare il setup del database di test eseguendo lo step 6. del procedimento sopra indicato.

Qualora il database di test debba essere utilizzato per testare le funzionalità dell'applicazione Go, è necessario modificare il file *.env* indicando *ica_test* nella variabile *DB_URL* ed eseguire un restart del relativo servizio *systemd*.

Questo switch del database deve essere eseguito solo in giorni o in orari in cui non interferisce con la normale operatività dell'applicazione (caricamento di *files* da parte degli utenti, remote upload, etc.).

3.7. Attivazione di una nuova azienda

L'attivazione di una nuova azienda è un'attività critica ai fini dell'operatività del sistema di backend per tutte le operazioni di caricamento *files*, gestione utenti e gestione *providers* che riguardano la nuova azienda. Di seguito si forniscono istruzioni dettagliate sia per il setup del database che dell'applicazione Go, rinviando ove necessario ad altre parti del presente documento. L'esempio non casuale è quello di una nuova azienda individuata con la sigla "RM1".

3.7.1. Configurazione del filesystem

La prima operazione da eseguire è la creazione della struttura di cartelle per l'azienda RM1, che deve essere eseguita sul server di backend. Il filesystem deve essere organizzato come rappresentato nell'immagine che segue.

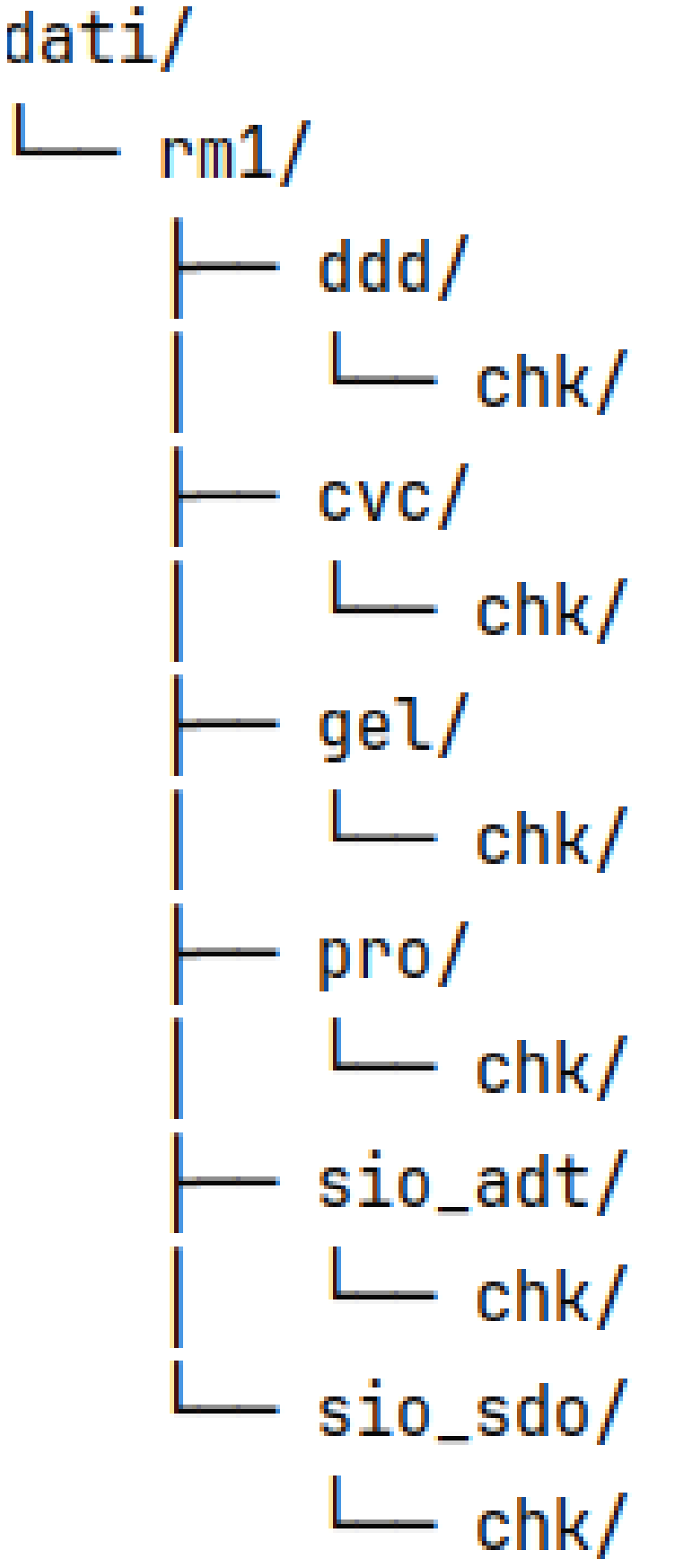

Fig. 34 - La struttura del filesystem per azienda

La creazione delle cartelle è eseguita con il comando bash che segue.

Creazione della struttura delle cartelle

```
nomos$> mkdir -p dati/rm1/{ddd/chk,cvc/chk,gel/chk,pro/chk,sio_adt/chk,sio_sdo/chk}
```

Dopo questa operazione il filesystem è pronto per ricevere i *files* caricati manualmente o da remoto e per destinarli alle cartelle di archiviazione secondo le regole della *naming convention*.

3.7.2. Configurazione dell'applicazione Go

- aggiunta flussi su flows.yml
- aggiunta azienda su config.yml

3.7.3. Configurazione del database

Esecuzione in 2 steps: database di test e database di produzione

Presupposti: disponibilità tabelle centri di costo e reparti + estrazioni flussi

- inserimento metadati in _dd_tables
- *schema_cloning*
- test caricamenti flussi

[1] gli *steps* 1. e 4. sono necessari solo se il database di test non è allineato con il database di produzione

Chapter 4. Operation management

Con la crescita del numero delle aziende e l'integrazione del *remote upload*, la gestione del backend non può prescindere da attività di pianificazione e monitoraggio.

A supporto di tali attività gli *operation data* forniscono importanti risorse informative. La pipeline crea una scia di dati sul processo che è disseminata in varie tabelle:

- la *flow_instance* che ha come chiave primaria il *job_id* e raccoglie le specifiche di ogni caricamento, sia esso manuale o remoto;
- la *job progress* che traccia i singoli passaggi della *flow_instance* nella pipeline;
- l'application logs, che tiene traccia degli errori (di sistema o di dominio) che possono comportare un'interruzion del processo;
- *dd_csv_files* che riporta il log dei caricamenti.

Queste tabelle sono tutte collegate attraverso il *job_id* e quindi forniscono la **tracciabilità** dei caricamenti in un contesto di struttura aziendale. Il modello dei dati che rappresenta questa architettura è di seguito riportato.

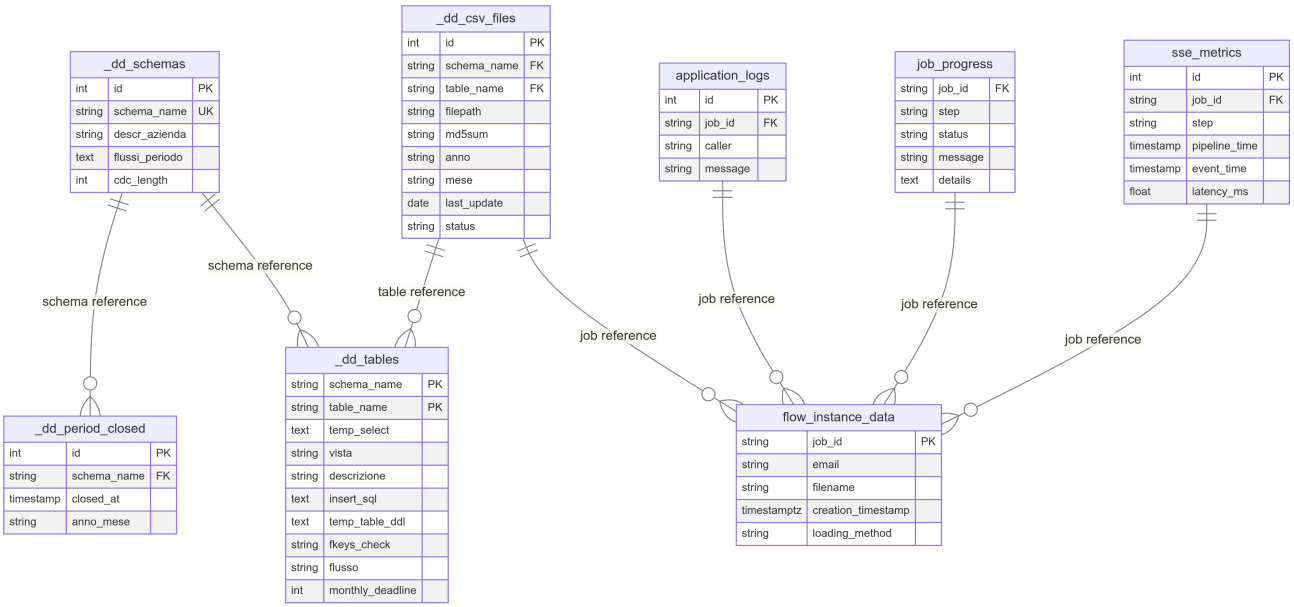

Fig. 35 - Il modello dei dati per il tracking dei caricamenti

Il potenziale informativo di questo modello si riscontra in vari livelli di analisi:

- le tempistiche delle operazioni di caricamento e delle chiusure mensili;
- la tracciabilità delle singole operazioni di caricamento e delle eventuali relative anomalie;
- l'analisi degli errori di sistema e di dominio per consentire il progressivo *enhancement* dell'applicazione.

Lo strumento disponibile per l'analisi dei dati è il *report generator*, che consente di definire viste

personalizzate sugli *operation data*. Parte di queste viste possono essere condivise con gli utenti *uploader* e *providers* per consentire loro di monitorare i propri caricamenti e di avere un feedback immediato in caso di errori. Altre viste, di livello sovra-ziendale, possono riportare al management Nomos un quadro complessivo dell'operatività del backend, con una lettura top-down fino al singolo caricamento.

Ovviamente l'operation management va oltre l'analisi dei dati: la sua funzione è l'ottimizzazione dei processi a supporto della qualità del servizio fornito. Questa funzione si esercita attraverso la pianificazione e il monitoraggio delle operazioni di caricamento e attraverso una standardizzazione della gestione *day-by-day* per quanto riguarda i rapporti con i referenti aziendali o con i *providers* su tematiche quali:

1. Tempi di consegna dei dati;
2. Modalità operative di recupero degli errori di caricamento che per motivi diversi richiedono la sostituzione del file;
3. Modalità operative di recupero degli *errori di dominio* nei dati, che includono:
    - la presenza di nuovi codici di unità operative o di centri di costo;
    - la presenza di nuovi codici di prodotto nei flussi della Farmacia;
    - presenza del delimitatore di colonna in un campo descrittivo del file csv;
4. Nuove aree da presidiare, ad esempio connesse all'attivazione e alla gestione *day-by-day* del remote upload.