Indice

1. Introduzione

Il seguente elaborato ha lo scopo di documentare la progettazione e lo sviluppo di una base di dati relazionale del DBMS PostgreSQL, ad opera degli studenti:

- Pocomento Antonio,
- Ruggiero Giulio,
- Simone Rampone Eugenio

del CdL in Informatica presso l'Università degli Studi di Napoli "Federico II".

È possibile consultare il progetto nel suo insieme, compresa la parte riguardante la programmazione Object Oriented, tramite il seguente URL: link GitHub (https://github.com/Antonio-Pocomento/SistemaAeroporto).

1.1. Descrizione del problema

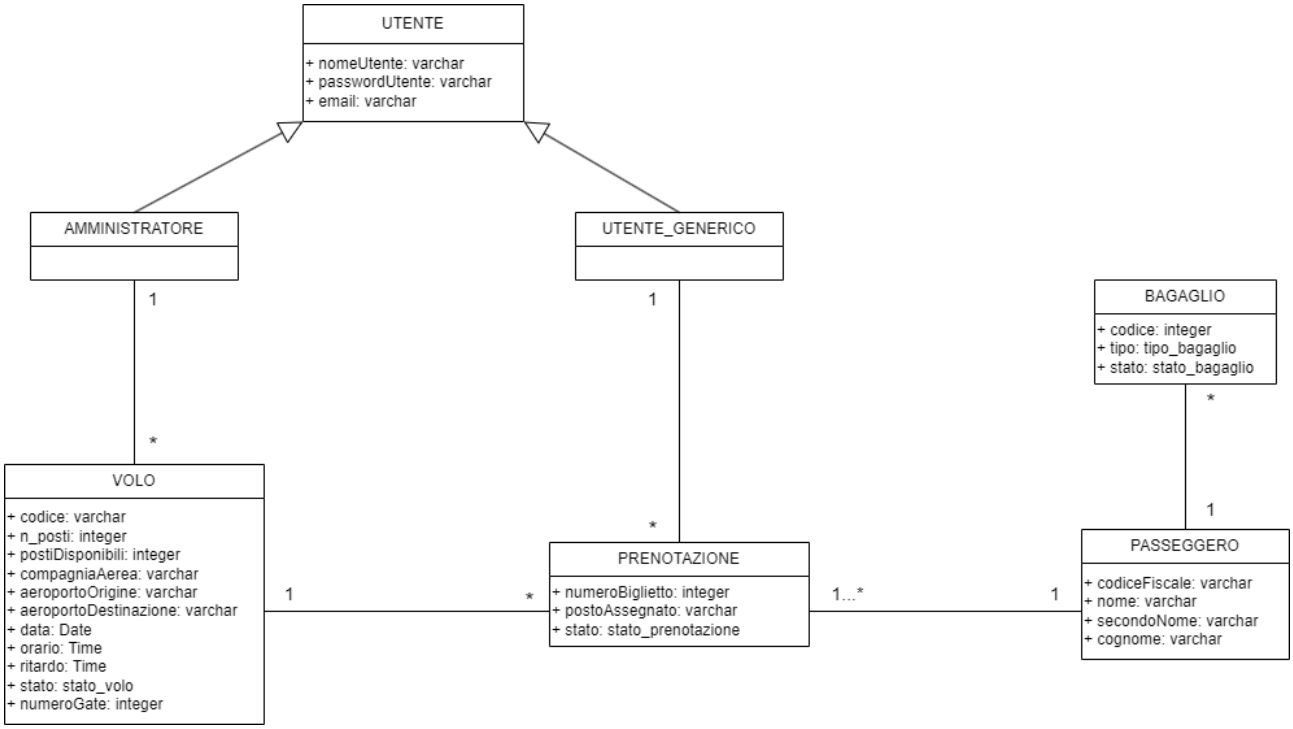
Verranno riportate progettazione e sviluppo di una base di dati relazionale che consenta di memorizzare le informazioni necessarie per il funzionamento di un sistema informativo per la gestione dell'aeroporto di Napoli.

Il sistema può essere utilizzato da utenti autenticati tramite una login (nome utente o email) e una password. Gli utenti sono suddivisi in due ruoli: utenti generici, che possono prenotare voli, e amministratori del sistema.
Il sistema gestisce i voli in arrivo e quelli in partenza. Ogni volo è caratterizzato da un codice univoco, la compagnia aerea, l'aeroporto di origine (per i voli in arrivo a Napoli) e quello di destinazione (per i voli in partenza da Napoli), la data del volo, l'orario previsto, l'eventuale ritardo e lo stato del volo (programmato, decollato, in ritardo, atterrato, cancellato).
Gli utenti generici possono effettuare prenotazioni per i voli programmati. Ogni prenotazione è legata a un volo e contiene informazioni come i dati del passeggero (che non deve necessariamente coincidere con il nome dell'utente che lo ha prenotato), il numero del biglietto, il posto assegnato e lo stato della prenotazione (confermata, in attesa, cancellata).
Il sistema gestisce anche i gate di imbarco (identificati da un numero), assegnandoli ai voli in partenza.
Un'altra funzione importante è il monitoraggio dei bagagli. Ogni bagaglio viene registrato nel sistema durante l'operazione di check-in, generando un codice univoco che consente il tracciamento. Durante il percorso, lo stato del bagaglio viene gestito manualmente dagli amministratori.

2. Progettazione Concettuale

In questo capitolo documentiamo la progettazione del database al suo livello di astrazione più alto. Partendo dall'analisi dei requisiti da soddisfare, si arriverà ad uno schema concettuale indipendente dalla struttura dei dati e dall'implementazione fisica degli stessi, rappresentato con un Class Diagram UML.

2.1. Class Diagram



2.2. Ristrutturazione del Class Diagram

Si procede alla ristrutturazione del Class Diagram, al fine di rendere quest'ultimo idoneo alla traduzione in schemi relazionali e di migliorarne l'efficienza. La ristrutturazione seguirà i seguenti punti:

• Analisi delle chiavi

• Analisi degli attributi derivati

• Analisi delle ridondanze

• Analisi degli attributi strutturati

• Analisi degli attributi a valore multiplo

• Analisi delle gerarchie di specializzazione

2.2.1. Analisi delle chiavi

Fortunatamente, ogni classe presenta un parametro che può essere utilizzato come chiave:

- per la classe **volo**, sarà utilizzato il codice;
- per la classe **prenotazione**, sarà utilizzato il numero del biglietto;
- per la classe **passeggero**, sarà utilizzato il suo codice fiscale;
- per la classe **bagaglio**, sarà utilizzato il codice;
- per la classe **utente**, si utilizzerà il nome utente.

NOTA: Anche se il nome utente rischia di essere modificato, anche più volte, si è deciso di sfruttare un identificatore naturale e semanticamente significativo, evitando colonne surrogate non necessarie.

2.2.2. Analisi degli attributi derivati

Per ottimizzare ulteriormente l'utilizzo delle risorse di calcolo, analizziamo gli eventuali attributi derivati, ovvero calcolabili da altri attributi delle entità.

Dato che non sono presenti attributi derivati, passiamo al punto successivo.

2.2.3. Analisi delle ridondanze

Analizziamo ora l'eventuale presenza di associazioni ridondanti tra le varie entità, in maniera tale da evitare incoerenze nella rappresentazione logica dei dati.

Anche qui, dato che non sono presenti ridondanze evidenti, passiamo al punto successivo.

2.2.4. Analisi degli attributi strutturati

Vanno ora analizzati e concettualmente corretti eventuali attributi strutturati presenti nelle entità. Questi infatti non sono logicamente rappresentabili all'interno di un DBMS, e vanno quindi eliminati e codificati in altro modo.

Visto che non sono presenti attributi strutturati, procediamo con il prossimo punto.

2.2.5. Analisi degli attributi a valore multiplo

Verifichiamo ora la presenza di eventuali attributi a valore multiplo, anch'essi non logicamente rappresentabili e quindi da eliminare nello schema concettuale ristrutturato.

Nuovamente non è presente un attributo di questo tipo, dunque proseguiamo con l'ultimo punto.

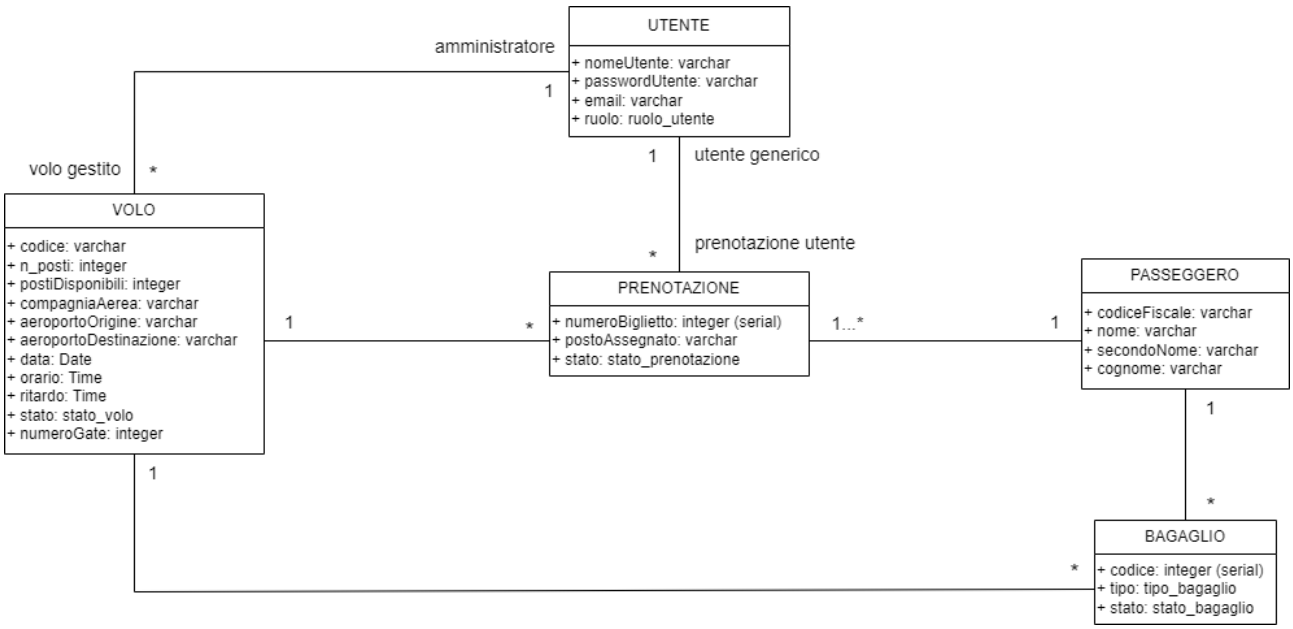
2.2.6. Analisi delle gerarchie di specializzazione

Infine andiamo a ristrutturare eventuali gerarchie di specializzazione, altro elemento non rappresentabile in un DBMS relazionale.

Osserviamo che la classe Utente presenta due specializzazioni: Utente Generico e Amministratore, ma in realtà non presentano alcun nuovo attributo rispetto alla classe Utente.

Eliminiamo dunque le due specializzazioni e introduciamo un attributo "ruolo" alla classe Utente, per poter distinguere tra generici utenti e amministratori.

2.3. Class Diagram Ristrutturato



Oltre a sostituire le specializzazioni con l'attributo "ruolo", si è scelto di creare una nuova associazione tra la classe Bagaglio e la classe Volo che, anche se ridondante

dal punto di vista logico, semplifica notevolmente delle operazioni nella progettazione fisica (eliminazione dei bagagli per un volo).

2.4. Dizionario delle Classi

Classe	Descrizione	Attributi
Utente	Descrive un utente che accede al sistema	**nomeUtente** (char[]) : nome scelto dall'utente. **passwordUtente** (char[]) : password scelta dall'utente. **email** (char[]) : email scelta dall'utente. **ruolo** (ruolo_utente) : indica se l'utente è un amministratore o un utente generico.
Volo	Descrive un volo presente nel sistema	**codice** (char[]) : codice del volo. **n_posti** (integer) : numero di posti del volo. **postiDisponibili** (integer) : numero di posti disponibili, quindi non occupati o prenotati, per il volo. **compagniaAerea** (char[]) : nome della compagnia aerea. **aeroportoOrigine** (char[]) : nome della città da cui parte il volo. **aeroportoDestinazione** (char[]) : nome della città di destinazione del volo. **data** (Date) : data di partenza del volo. **orario** (Time) : orario di partenza del volo. **ritardo** (Time) : eventuale ritardo previsto per il volo. **stato** (stato_volo) : attuale stato del volo. **numeroGate** (integer) : numero del gate di stazionamento del volo.
Prenotazione	Descrive una prenotazione effettuata da un utente	**numeroBiglietto** (integer) : numero univoco che identifica il biglietto. **postoAssegnato** (char[]) : posto assegnato per il volo. **stato** (stato_prenotazione) : stato attuale della prenotazione
Passeggero	Descrive un passeggero prenotato per un volo	**codiceFiscale** (char[]) : codice fiscale del passeggero. **nome** (char[]) : nome del passeggero. **secondoNome** (char[]) : eventuale secondo nome del passeggero.

		cognome (char[]) : cognome del passeggero.
Bagaglio	Descrive un bagaglio trasportato da un passeggero	**codice** (integer) : codice univoco che identifica il bagaglio. **tipo** (tipo_bagaglio) : tipo del bagaglio. **stato** (stato_bagaglio) : stato del bagaglio.

2.5. Dizionario delle Associazioni

Nome	Descrizione	Classi coinvolte
Gestione Volo	Esprime la gestione di un volo da parte di un amministratore	Utente [1] : ogni volo deve essere gestito da un amministratore. Volo [*] : ogni amministratore può gestire 0,1 o più voli.
Prenotazione Effettuata	Esprime l'effettuata prenotazione da parte dell'utente	Utente [1] : ogni prenotazione è associata ad un utente. Prenotazione [*] : ogni utente può effettuare 0,1 o più prenotazioni.
Volo Prenotato	Esprime il volo per cui è stata effettuata la prenotazione	Volo [1] : ogni prenotazione è associata ad un volo. Prenotazione [*] : ogni volo può essere associato a 0,1 o più prenotazioni.
Passeggero Prenotato	Esprime il passeggero a cui si riferisce la prenotazione effettuata dall'utente	Passeggero [1] : ogni prenotazione è associata ad un passeggero. Prenotazione [1…*] : ogni passeggero può prenotarsi per uno o più voli (non sono considerati nella base di dati i passeggeri senza prenotazioni).
Bagaglio Passeggero	Esprime il bagaglio trasportato dal passeggero	Passeggero [1] : ogni bagaglio è trasportato da un passeggero. Bagaglio [*] : ogni passeggero può

		trasportare 0,1 o più bagagli.
Bagaglio Su Volo	Esprime il volo che trasporta il bagaglio	Volo [1] : ogni bagaglio è trasportato (o sarà trasportato) da un volo. Bagaglio [*] : ogni volo può trasportare 0,1 o più bagagli

2.6. Dizionario dei Vincoli

Nome	Descrizione
chk_caratteri_validi	Verifica che il nome utente e la password non presentino caratteri non permessi, come lo spazio.
chk_email	Verifica che il testo inserito corrisponda allo schema tipico di un'email.
lunghezza_campi_utente	Verifica che il nome utente inserito abbia almeno 3 caratteri, e che la password inserita abbia almeno 8 caratteri
chk_posto	Verifica che il posto inserito corrisponda al seguente schema: Numero di fila (da 1 a 99) + Lettera di posto (da A a K, saltando la I).
chk_codice_volo	Verifica che il codice inserito corrisponda al seguente schema: Codice della compagnia (due lettere o una lettera più una cifra) + codice del volo (da 1 a 4 cifre) + un eventuale ultima lettera
lunghezza_campi_volo	Verifica che la compagnia aerea, l'aeroporto di origine e l'aeroporto di destinazione abbiano almeno un carattere.
ritardo_volo	Verifica che un volo, se presenta un ritardo, abbia come stato 'IN_RITARDO', altrimenti, se il campo del ritardo è [null], lo stato del volo deve essere diverso da 'IN_RITARDO'.

volo_check_aeroporti	Verifica che almeno uno dei due aeroporti (di destinazione o di origine) sia quello di Napoli, e che i due aeroporti siano diversi.
volo_check_posti	Verifica che i posti del volo siano maggiori di zero e che i posti disponibili siano maggiori o pari a 0 e minori o uguali ai posti del volo.
gate_partenza_volo	Verifica che un volo in partenza abbia un numero di gate, mentre un volo in arrivo abbia un valore [null].
chk_campi_solo_lettere	Verifica che il nome, secondo nome (se presente) e il cognome del passeggero contengano solo lettere.
chk_codice_fiscale	Verifica che il codice fiscale inserito rispetti il seguente schema: sequenza di 6 caratteri, 2 interi, 1 carattere, 2 interi, 1 carattere, 3 interi, 1 carattere
lunghezza_campi_passeggero	Verifica che il nome, secondo nome (se presente) e il cognome del passeggero abbiano almeno un carattere

3. Progettazione Logica

In questo capitolo tratteremo la seconda fase della progettazione, scendendo ad un livello di astrazione più basso rispetto al precedente. Lo schema concettuale verrà tradotto, anche grazie alla predisposizione conseguente la ristrutturazione, in uno schema logico, questa volta dipendente dalla struttura dei dati prescelta, nello specifico quella relazionale pura.

3.1. Schema Logico

Di seguito è riportato lo schema logico della base di dati. Al suo interno, le chiavi primarie sono indicate con una sottolineatura singola mentre le chiavi esterne con una sottolineatura doppia.

- **Utente**(nomeUtente, passwordUtente, email, ruolo)
- **Passeggero**(codiceFiscale, nome, secondoNome, cognome)

- **Volo**(codice, nPosti, postiDisponibili, compagniaAerea, aeroportoOrigine, aeroportoDestinazione, data, orario, ritardo, stato, numeroGate, amministratore)
  amministratore → Utente.nomeUtente

- **Bagaglio**(codice, tipo, stato, codiceFiscalePasseggero, codiceVolo)
  codiceFiscalePasseggero → Passeggero.codiceFiscale
  codiceVolo → Volo.codice

- **Prenotazione**(numeroBiglietto, postoAssegnato, stato, codiceVolo, nomeUtente, codiceFiscale)
  codiceVolo → Volo.codice
  nomeUtente → Utente.nomeUtente
  codiceFiscale → Passeggero.codiceFiscale

4. Progettazione Fisica: Tabelle

In questo capitolo verrà riportata l'implementazione dello schema logico sopra descritto nel DBMS PostgreSQL.

4.1. Tabella Utente

```
CREATE TABLE IF NOT EXISTS utente (
    nomeutente VARCHAR(255) NOT NULL,
    passwordutente VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    ruolo ruolo_utente NOT NULL DEFAULT 'generico',

    CONSTRAINT pk_nomeutente PRIMARY KEY (nomeutente),
    CONSTRAINT uq_email UNIQUE (email),
    CONSTRAINT chk_caratteri_validi CHECK (
        nomeutente ~ '^[A-Za-z0-9?!@#$%^&*()_+=-]+$'
        AND passwordutente ~ '^[A-Za-z0-9?!@#$%^&*()_+=-]+$'
    ),
    CONSTRAINT chk_email CHECK (
        email ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'
    ),
    CONSTRAINT lunghezza_campi_utente CHECK (
        length(trim(nomeutente)) >= 3
        AND length(trim(passwordutente)) >= 8
    )
);
```

4.2. Tabella Passeggero

```
CREATE TABLE IF NOT EXISTS passeggero (
    codice_fiscale VARCHAR NOT NULL,
    nome VARCHAR NOT NULL,
    secondo_nome VARCHAR,
    cognome VARCHAR NOT NULL,

    CONSTRAINT passeggero_pkey PRIMARY KEY (codice_fiscale),

    CONSTRAINT chk_campi_solo_lettere CHECK (
        cognome ~ '^[[:alpha:]'' ]+$'
        AND nome ~ '^[[:alpha:]'' ]+$'
        AND (secondo_nome IS NULL OR secondo_nome ~ '^[[:alpha:]'' ]+$')
    ),

    CONSTRAINT chk_codice_fiscale CHECK (
        codice_fiscale ~ '^[A-Z]{6}[0-9]{2}[A-Z][0-9]{2}[A-Z][0-9]{3}[A-
Z]$'
    ),

    CONSTRAINT lunghezza_campi_passeggero CHECK (
        length(trim(nome)) > 0
        AND length(trim(cognome)) > 0
        AND (secondo_nome IS NULL OR length(trim(secondo_nome)) > 0)
    )
);
```

4.3. Tabella Bagaglio

```
CREATE TABLE IF NOT EXISTS bagaglio (
    stato stato_bagaglio NOT NULL DEFAULT 'REGISTRATO',
    tipo tipo_bagaglio NOT NULL,
    codice_fiscale_passeggero VARCHAR NOT NULL,
    codice INTEGER NOT NULL DEFAULT nextval('bagaglio_codice_seq'),
    codice_volo VARCHAR(100) NOT NULL,

    CONSTRAINT bagaglio_pkey PRIMARY KEY (codice),

    CONSTRAINT fk_codice_fiscale FOREIGN KEY (codice_fiscale_passeggero)
        REFERENCES passeggero (codice_fiscale)
        ON DELETE CASCADE,

    CONSTRAINT fk_codice_volo FOREIGN KEY (codice_volo)
        REFERENCES volo (codice)
        ON DELETE CASCADE
);
```

4.4. Tabella Prenotazione

```
CREATE TABLE IF NOT EXISTS prenotazione (
    numero_biglietto INTEGER NOT NULL DEFAULT
nextval('prenotazione_numerobiglietto_seq'),
    posto_assegnato VARCHAR NOT NULL,
    stato stato_prenotazione NOT NULL DEFAULT 'IN_ATTESA',
    codice_volo VARCHAR NOT NULL,
    nome_utente VARCHAR NOT NULL,
    codice_fiscale VARCHAR NOT NULL,

    CONSTRAINT prenotazione_pkey PRIMARY KEY (numero_biglietto),

    CONSTRAINT fk_codice_fiscale FOREIGN KEY (codice_fiscale)
        REFERENCES passeggero (codice_fiscale)
        ON DELETE CASCADE,

    CONSTRAINT fk_nome_utente FOREIGN KEY (nome_utente)
        REFERENCES utente (nomeutente)
        ON UPDATE CASCADE
        ON DELETE CASCADE,

    CONSTRAINT fk_volo FOREIGN KEY (codice_volo)
        REFERENCES volo (codice)
        ON DELETE CASCADE,

    CONSTRAINT chk_posto CHECK (
        posto_assegnato ~ '^[1-9][0-9]?[A-HJ-K]$'
    )
);
```

4.5. Tabella Volo

```
CREATE TABLE IF NOT EXISTS volo (
    codice VARCHAR(100) NOT NULL,
    n_posti INTEGER NOT NULL,
    posti_disponibili INTEGER NOT NULL,
    compagnia_aerea VARCHAR(100) NOT NULL,
    aeroporto_origine VARCHAR(100) NOT NULL,
    aeroporto_destinazione VARCHAR(100) NOT NULL,
    data DATE NOT NULL,
    orario TIME NOT NULL,
    ritardo TIME,
    stato stato_volo NOT NULL DEFAULT 'PROGRAMMATO',
    numero_gate INTEGER,
    amministratore VARCHAR(100) NOT NULL,

    CONSTRAINT volo_pkey PRIMARY KEY (codice),

    CONSTRAINT fk_amministratore FOREIGN KEY (amministratore)
        REFERENCES utente (nomeutente)
        ON UPDATE CASCADE
        ON DELETE RESTRICT,

    CONSTRAINT chk_codice_volo CHECK (
        codice ~ '^[A-Z0-9]{2}[0-9]{1,4}[A-Z]?$'
    ),

    CONSTRAINT gate_partenza_volo CHECK (
        (numero_gate IS NULL AND aeroporto_destinazione = 'Napoli')
        OR (numero_gate IS NOT NULL AND aeroporto_origine = 'Napoli')
    ),

    CONSTRAINT lunghezza_campi_volo CHECK (
        length(trim(aeroporto_origine)) > 0
        AND length(trim(aeroporto_destinazione)) > 0
        AND length(trim(compagnia_aerea)) > 0
    ),

    CONSTRAINT ritardo_volo CHECK (
        (ritardo IS NULL AND stato <> 'IN_RITARDO')
        OR (ritardo IS NOT NULL AND stato = 'IN_RITARDO')
    ),

    CONSTRAINT volo_check_aeroporti CHECK (
        (aeroporto_origine = 'Napoli' OR aeroporto_destinazione =
'Napoli')
        AND aeroporto_origine <> aeroporto_destinazione
    ),

    CONSTRAINT volo_check_posti CHECK (
```

```
        n_posti > 0
        AND posti_disponibili >= 0
        AND posti_disponibili <= n_posti
    )
);
```

5. Progettazione Fisica: Trigger

Di seguito sono riportate le definizioni dei trigger e delle corrispettive trigger function.

5.1. Check_Amministratore()

Trigger per la tabella volo.

```
CREATE OR REPLACE FUNCTION public.check_amministratore()
RETURNS trigger
LANGUAGE plpgsql
AS $$
DECLARE
    v_ruolo TEXT;
BEGIN
    -- Recupera il ruolo dell'utente
    SELECT ruolo
    INTO v_ruolo
    FROM utente
    WHERE nomeutente = NEW.amministratore;

    -- Controlla se l'utente esiste
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Utente "%" non esistente.', NEW.amministratore;
    END IF;

    -- Verifica che sia un amministratore
    IF v_ruolo <> 'amministratore' THEN
        RAISE EXCEPTION 'Un utente generico non può gestire un volo.';
    END IF;

    RETURN NEW;
END;
$$;

CREATE OR REPLACE TRIGGER trg_check_amministratore
    BEFORE INSERT OR UPDATE
    ON public.volo
    FOR EACH ROW
    EXECUTE FUNCTION public.check_amministratore();
```

5.2. Check_Date()

Trigger per la tabella volo.

```
CREATE OR REPLACE FUNCTION public.check_date()
RETURNS trigger
LANGUAGE plpgsql
AS $$
BEGIN
    -- Sull'inserimento: sempre verifica
    IF TG_OP = 'INSERT' THEN
        IF NEW.data < current_date THEN
            RAISE EXCEPTION 'La data del volo deve essere oggi o
successiva.';
        END IF;
    END IF;

    -- Sugli update: verifica solo se la data cambia
    IF TG_OP = 'UPDATE' THEN
        IF NEW.data <> OLD.data THEN
            IF NEW.data < current_date THEN
                RAISE EXCEPTION 'Non puoi modificare la data del volo a
una data passata.';
            END IF;
        END IF;
    END IF;

    RETURN NEW;
END;
$$;

CREATE OR REPLACE TRIGGER trg_check_date
    BEFORE INSERT OR UPDATE
    ON public.volo
    FOR EACH ROW
    EXECUTE FUNCTION public.check_date();
```

5.3. Delete_Bagagli()

Trigger per la tabella prenotazione.

```
CREATE OR REPLACE FUNCTION public.delete_bagagli()
RETURNS trigger
LANGUAGE plpgsql
AS $$
BEGIN
    -- Se lo stato NON sta diventando 'CANCELLATA', non fare niente
    IF NEW.stato <> 'CANCELLATA' THEN
        RETURN NEW;
    END IF;

    -- Elimina i bagagli di questo passeggero per questo volo
    DELETE FROM bagaglio
    WHERE codice_fiscale_passeggero = NEW.codice_fiscale
      AND codice_volo = NEW.codice_volo;

    RETURN NEW;
END;
$$;


CREATE OR REPLACE TRIGGER trg_delete_bagagli
    AFTER UPDATE
    ON public.prenotazione
    FOR EACH ROW
    WHEN (old.stato IS DISTINCT FROM new.stato AND new.stato =
'CANCELLATA'::stato_prenotazione)
    EXECUTE FUNCTION public.delete_bagagli();
```

5.4. Aggiorna_Posti_Disponibili()

Trigger per la tabella prenotazione.

```
CREATE OR REPLACE FUNCTION public.aggiorna_posti_disponibili()
RETURNS trigger
LANGUAGE plpgsql
AS $$
BEGIN
    -- Caso INSERT: decrementa posti_disponibili se prenotazione attiva
    IF TG_OP = 'INSERT' THEN
        IF NEW.stato <> 'CANCELLATA' THEN
            UPDATE volo
            SET posti_disponibili = posti_disponibili - 1
            WHERE codice = NEW.codice_volo;
        END IF;

    -- Caso DELETE: incrementa posti_disponibili se prenotazione attiva
    ELSIF TG_OP = 'DELETE' THEN
        IF OLD.stato <> 'CANCELLATA' THEN
            UPDATE volo
            SET posti_disponibili = posti_disponibili + 1
            WHERE codice = OLD.codice_volo;
        END IF;

    -- Caso UPDATE: controlla variazioni di stato e codice_volo
    ELSIF TG_OP = 'UPDATE' THEN
        -- Se lo stato passa da attivo a cancellato: incremento posti
        IF OLD.stato <> 'CANCELLATA' AND NEW.stato = 'CANCELLATA' THEN
            UPDATE volo
            SET posti_disponibili = posti_disponibili + 1
            WHERE codice = NEW.codice_volo;

        -- Se cambia il volo mantenendo stato attivo: aggiorna posti su
entrambi i voli
        ELSIF OLD.codice_volo <> NEW.codice_volo THEN
            IF OLD.stato <> 'CANCELLATA' THEN
                UPDATE volo
                SET posti_disponibili = posti_disponibili + 1
                WHERE codice = OLD.codice_volo;
            END IF;

            IF NEW.stato <> 'CANCELLATA' THEN
                UPDATE volo
                SET posti_disponibili = posti_disponibili - 1
                WHERE codice = NEW.codice_volo;
            END IF;
        END IF;
    END IF;
```

```
    RETURN NEW;
END;
$$;

CREATE OR REPLACE TRIGGER trg_aggiorna_posti_disponibili
    AFTER INSERT OR DELETE OR UPDATE
    ON public.prenotazione
    FOR EACH ROW
    EXECUTE FUNCTION public.aggiorna_posti_disponibili();
```

5.5. Blocca_Update()

Trigger per la tabella prenotazione.

```
CREATE OR REPLACE FUNCTION public.blocca_update()
RETURNS trigger
LANGUAGE plpgsql
AS $$
BEGIN
  IF OLD.stato = 'CANCELLATA' THEN
    RAISE EXCEPTION 'Modifiche non permesse su prenotazioni cancellate';
  END IF;
  RETURN NEW;
END;
$$;

CREATE OR REPLACE TRIGGER trg_blocca_update
    BEFORE UPDATE
    ON public.prenotazione
    FOR EACH ROW
    EXECUTE FUNCTION public.blocca_update();
```

5.6. Check_Validita_Volo()

Trigger per la tabella prenotazione.

```
CREATE OR REPLACE FUNCTION public.check_validita_volo()
RETURNS trigger
LANGUAGE plpgsql
AS $$
DECLARE
    v_stato stato_volo;
    v_posti integer;
    v_origine text;
BEGIN
    -- Recupera informazioni sul volo
    SELECT stato, posti_disponibili, aeroporto_origine
    INTO v_stato, v_posti, v_origine
    FROM volo
    WHERE codice = NEW.codice_volo;

    -- Controlla se il volo esiste
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Il volo "%" non esiste.', NEW.codice_volo;
    END IF;

    -- Volo non cancellato
    IF v_stato = 'CANCELLATO' THEN
        RAISE EXCEPTION 'Il volo "%" è stato cancellato.',
NEW.codice_volo;
    END IF;

    -- Volo non decollato
    IF v_stato = 'DECOLLATO' THEN
        RAISE EXCEPTION 'Il volo "%" è già decollato.', NEW.codice_volo;
    END IF;

    -- Posti disponibili
    IF v_posti <= 0 THEN
        RAISE EXCEPTION 'Nessun posto disponibile sul volo "%".',
NEW.codice_volo;
    END IF;

    -- Aeroporto di partenza
    IF v_origine <> 'Napoli' THEN
        RAISE EXCEPTION 'Il volo "%" non parte da Napoli.',
NEW.codice_volo;
    END IF;

    RETURN NEW;
END;
$$;
```

```
CREATE OR REPLACE TRIGGER trg_check_validita_volo
    BEFORE INSERT
    ON public.prenotazione
    FOR EACH ROW
    EXECUTE FUNCTION public.check_validita_volo();
```

5.7. Check_Doppia_Prenotazione()

Trigger per la tabella prenotazione.

```
CREATE OR REPLACE FUNCTION public.check_doppia_prenotazione()
RETURNS trigger
LANGUAGE plpgsql
AS $$
BEGIN
  IF EXISTS (
    SELECT 1
    FROM prenotazione
    WHERE codice_volo = NEW.codice_volo
      AND codice_fiscale = NEW.codice_fiscale
      AND stato <> 'CANCELLATA'
      AND (TG_OP = 'INSERT' OR numero_biglietto <> NEW.numero_biglietto)
  )
  AND NEW.stato <> 'CANCELLATA' THEN
    RAISE EXCEPTION 'Esiste già una prenotazione attiva per questo
passeggero e questo volo';
  END IF;

  RETURN NEW;
END;
$$;

CREATE OR REPLACE TRIGGER trg_check_doppia_prenotazione
    BEFORE INSERT OR UPDATE
    ON public.prenotazione
    FOR EACH ROW
    EXECUTE FUNCTION public.check_doppia_prenotazione();
```

5.8. Check_Unique_Posto()

Trigger per la tabella prenotazione.

```
CREATE OR REPLACE FUNCTION public.check_unique_posto()
RETURNS trigger
LANGUAGE plpgsql
AS $$
DECLARE
    n INT;
BEGIN
    SELECT COUNT(*) INTO n
    FROM prenotazione
    WHERE codice_volo = NEW.codice_volo
      AND posto_assegnato = NEW.posto_assegnato
      AND stato <> 'CANCELLATA'
      AND numero_biglietto <> NEW.numero_biglietto;

    IF n > 0 AND NEW.stato <> 'CANCELLATA' THEN
        RAISE EXCEPTION 'Posto "%" sul volo "%" è già prenotato da un
altro utente attivo.',
            NEW.posto_assegnato, NEW.codice_volo;
    END IF;

    RETURN NEW;
END;
$$;


CREATE OR REPLACE TRIGGER trg_check_unique_posto
    BEFORE INSERT OR UPDATE
    ON public.prenotazione
    FOR EACH ROW
    EXECUTE FUNCTION public.check_unique_posto();
```

5.9. Check_Utente_Non_Amministratore()

Trigger per la tabella prenotazione.

```
CREATE OR REPLACE FUNCTION public.check_utente_non_amministratore()
RETURNS trigger
LANGUAGE plpgsql
AS $$
DECLARE
    v_ruolo text;
BEGIN
    -- Recupera il ruolo dell'utente
    SELECT ruolo
    INTO v_ruolo
    FROM utente
    WHERE nomeutente = NEW.nome_utente;

    -- Controlla se l'utente esiste
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Utente "%" non esistente.', NEW.nome_utente;
    END IF;

    -- Verifica che NON sia un amministratore
    IF v_ruolo = 'amministratore' THEN
        RAISE EXCEPTION 'Un amministratore non può effettuare
prenotazioni.';
    END IF;

    RETURN NEW;
END;
$$;


CREATE OR REPLACE TRIGGER trg_check_utente_non_amministratore
    BEFORE INSERT OR UPDATE
    ON public.prenotazione
    FOR EACH ROW
    EXECUTE FUNCTION public.check_utente_non_amministratore();
```