Basi di Dati

Progetto A.A. 2021/2022

SISTEMA DI GESTIONE DI UNA PIZZERIA

0279262

Francesca Pavone Salafia

Indice

1. Descrizione del Minimondo

Si vuole progettare il backend di un sistema informativo per la gestione dell'operatività di una pizzeria. In tale pizzeria è di interesse tenere traccia dei tavoli disponibili ed assegnati, dei camerieri associati ai tavoli, dei pizzaioli che preparano le pizze, del barista, del manager. Ciascuno dei lavoratori della pizzeria ha differenti mansioni e può effettuare operazioni differenti all'interno del sistema.

All'ingresso di un cliente, il manager lo riceve e lo registra, segnando nome, cognome e numero di commensali, assegnando un tavolo disponibile in grado di ospitarli tutti.

Un cameriere ha sempre la possibilità di visualizzare quali tavoli a lui assegnati sono occupati e quali sono stati serviti. Al momento di prendere l'ordine, il cameriere registra la comanda. Parte delle ordinazioni sono espletate dal barista, parte dal pizzaiolo.

Barista, pizzaiolo hanno sempre la possibilità di visualizzare cosa debbono preparare, in ordine di ricezione della comanda. Quando hanno preparato una bevanda o una pizza, il cameriere può visualizzare cosa è pronto (in relazione agli ordini) e sapere cosa deve consegnare a quale tavolo.

La pizzeria opera 24/7, ma per motivi di risparmio, in alcuni giorni sono disponibili un numero differente di camerieri e vengono utilizzati un numero differente di tavoli. Il manager può definire quali camerieri lavorano in quali turni e quali tavoli sono utilizzati in quali turni. Il menu è unico per tutti i turni e definito dal manager, con i rispettivi prezzi. Nel menu è necessario anche prevedere aggiunte per le pizze (ad esempio, un cliente potrebbe voler aggiungere del tonno ad una pizza quattro formaggi), con i relativi costi.

Allo stesso modo, il manager ha la possibilità di tenere traccia delle disponibilità dei singoli prodotti. In questo modo, se viene ordinato ad un cameriere da un cliente un prodotto che non è disponibile, questo non potrà essere aggiunto all'ordine.

Il manager ha la possibilità di stampare lo scontrino di un ordine. Inoltre, per motivi statistici, ha la possibilità di visualizzare le entrate giornaliere e/o mensili.

2. Analisi dei Requisiti

Identificazione dei termini ambigui e correzioni possibili

Linea	Termine	Nuovo termine	Motivo correzione
2	Assegnati	Occupati	Il termine “occupato” rende più chiaro il concetto di non disponibilità di un tavolo che essendo già occupato da un gruppo di commensali non sarà disponibile
3	Dei camerieri associati ai tavoli	Dei camerieri e della loro associazione ai tavoli	In questo modo sottolineo il fatto che bisogna tener traccia dei camerieri anche se essi non sono associati ad un tavolo in un determinato momento
3	Dei pizzaioli che preparano le pizze	Dei pizzaioli	La parte “che preparano le pizze” è superflua perché non devo tenere traccia per ogni pizza di quale pizzaiolo l’ha preparata
6	All’ingresso di un cliente, il manager lo riceve e lo registra	All’ingresso di un gruppo di commensali, il manager li riceve e registra uno di loro	Poiché al momento dell’arrivo il manager non riceve solo il cliente che andrà a registrare ma riceve l’intero gruppo di commensali e in seguito ne registra uno solo di essi come “rappresentante” del gruppo
8	Assegnati	Associati	Per coerenza utilizzo lo stesso termine utilizzato alla riga 3 per esprimere l’associazione dei camerieri ai tavoli
9	Quali sono stati serviti	In quali tavoli le relative comande da loro registrate sono state completamente servite	In questo modo è reso più chiaro il concetto di “tavolo servito”, che dipende quindi dallo stato della comanda relativa a quel determinato tavolo
10	Ordinazioni	Comande	Per non creare ambiguità dato che barista e pizzaiolo andranno a “consultare” la comanda che è stata registrata dal cameriere al momento dell’ordine
14	Consegnare	Servire	Il termine “servire” è più adatto all’ambito della ristorazione
23, 24	Ordine	Comanda	Alla riga 24 non utilizzo ne il termine “ordine” ne il termine “ordinazione” poiché il concetto di scontrino è legato alla comanda e non all’ordinazione/ordine che è inteso come la richiesta da parte dei commensali di aggiungere dei prodotti alla comanda (termine usato per coerenza anche alla riga 23)

Specifica disambiguata

All’ingresso di un gruppo di commensali, il manager li riceve e registra uno di loro, segnando nome, cognome e numero di commensali, assegnando un tavolo disponibile in grado di ospitarli tutti.
Un cameriere ha sempre la possibilità di visualizzare quali tavoli a lui associati sono occupati e in quali tavoli le relative comande da loro registrate sono state completamente servite. Al momento di prendere l’ordine, il cameriere registra la comanda. Parte delle comande sono espletate dal barista, parte dal pizzaiolo.
Barista, pizzaiolo hanno sempre la possibilità di visualizzare cosa debbono preparare, in ordine di ricezione della comanda. Quando hanno preparato una bevanda o una pizza, il cameriere può visualizzare cosa è pronto (in relazione agli ordini) e sapere cosa deve servire a quale tavolo.
La pizzeria opera 24/7, ma per motivi di risparmio, in alcuni giorni sono disponibili un numero differente di camerieri e vengono utilizzati un numero differente di tavoli. Il manager può definire quali camerieri lavorano in quali turni e quali tavoli sono utilizzati in quali turni. Il menu è unico per tutti i turni e definito dal manager, con i rispettivi prezzi. Nel menu è necessario anche prevedere aggiunte per le pizze (ad esempio, un cliente potrebbe voler aggiungere del tonno ad una pizza quattro formaggi), con i relativi costi.
Allo stesso modo, il manager ha la possibilità di tenere traccia delle disponibilità dei singoli prodotti. In questo modo, se viene ordinato ad un cameriere da un cliente un prodotto che non è disponibile, questo non potrà essere aggiunto alla comanda.
Il manager ha la possibilità di stampare lo scontrino di una comanda. Inoltre, per motivi statistici, ha la possibilità di visualizzare le entrate giornaliere e/o mensili.

Glossario dei Termini

Termine	Descrizione	Collegamenti
Lavoratore pizzeria	I lavoratori della pizzeria si distinguono in 4 tipi: cameriere, pizzaiolo, barista e manager.	Cameriere, Pizzaiolo, Barista, Manager
Cameriere	Lavoratore della pizzeria incaricato di prendere le ordinazioni e servire i tavoli a lui associati	Comanda, Tavolo, Turno
Manager	Lavoratore della pizzeria che si occupa dell'accoglienza dei clienti, dell'organizzazione dei turni, della verifica sulla disponibilità dei prodotti, della stampa degli scontrini e della visualizzazione dei resoconti giornalieri e/o mensili	Cliente, Turno, Prodotto, Scontrino
Comanda	La comanda viene registra quando i clienti effettuano l'ordinazione delle pizze (con eventuali aggiunte) e delle bevande tramite il cameriere associato al loro tavolo.	Tavolo, Scontrino, Cameriere, Pizza, Bevanda
Tavolo	I tavoli che il ristorante mette a disposizione per far accomodare i clienti e poterli servire.	Turni, Cliente, Cameriere, Comanda
Cliente	Il cliente è un componente del gruppo di commensali e viene utilizzato il suo nome e cognome per la registrazione.	Tavolo
Scontrino	Lo scontrino viene emanato per ogni tavolo (in riferimento alla comanda), e contiene il prezzo totale di essa	Comanda
Turno	Fascia oraria in cui lavora un gruppo di camerieri e viene utilizzati un insieme di tavoli	Cameriere, Tavolo
Menu	Il menu è costituito dall'insieme di tutti i prodotti	Prodotti
Prodotto	I prodotti sono di 3 tipi: pizze, bevande e ingredienti (aggiunte)	Ingrediente, Pizza, Bevanda
Aggiunta	Al momento dell'ordinazione un cliente può richiedere delle aggiunte sulla propria pizza, tutti gli ingredienti usati come condimento per le pizze possono essere usati anche come aggiunta	Pizza, Comanda
Pizza	Prodotto che può essere ordinato dai clienti della pizzeria, eventualmente applicando delle aggiunte su di esso. Ogni pizza è inoltre condita con diversi ingredienti.	Prodotto, Comanda, Aggiunta, Ingrediente
Bevanda	Le bevande messe a disposizione dalla pizzeria sono bevande industriali già pronte quindi non si tiene traccia degli ingredienti che la compongono	Prodotto, Comanda

Raggruppamento dei requisiti in insiemi omogenei

Frasi di carattere generale
Si vuole progettare il backend di un sistema informativo per la gestione dell'operatività di una pizzeria.
Ciascuno dei lavoratori della pizzeria ha differenti mansioni e può effettuare operazioni differenti all'interno del sistema.
La pizzeria opera 24/7

Frasi relative ai camerieri
In tale pizzeria è di interesse tenere traccia … dei camerieri e della loro associazione ai tavoli
Un cameriere ha sempre la possibilità di visualizzare quali tavoli a lui associati sono occupati e in quali tavoli le relative comande da lui registrate sono state completamente servite
Al momento di prendere l'ordinazione, il cameriere registra la comanda
Il cameriere può visualizzare cosa è pronto (in relazione agli ordini) e sapere cosa deve servire a quale tavolo
In alcuni giorni sono disponibili un numero differente di camerieri
Il manager può definire quali camerieri lavorano in quali turni

Frasi relative al manager
All'ingresso di un gruppo di commensali, il manager li riceve e registra uno di loro, segnando nome, cognome e numero di commensali, assegnando un tavolo disponibile in grado di ospitarli tutti
Il manager può definire quali camerieri lavorano in quali turni e quali tavoli sono utilizzati in quali turni
Il menu è ... definito dal manager, con i rispettivi prezzi
Il manager ha la possibilità di tenere traccia delle disponibilità dei singoli prodotti
Il manager ha la possibilità di stampare lo scontrino di una comanda
Per motivi statistici, ha la possibilità di visualizzare le entrate giornaliere e/o mensili

Frasi relative al pizzaiolo
Parte delle comande sono espletate ... dal pizzaiolo
Ha sempre la possibilità di visualizzare cosa deve preparare, in ordine di ricezione della comanda
Quando ha preparato una pizza, il cameriere può vedere cosa è pronto

Frasi relative al barista
Parte delle comande sono espletate dal barista
Ha sempre la possibilità di visualizzare cosa deve preparare, in ordine di ricezione della comanda
Quando ha preparato una bevanda, il cameriere può vedere cosa è pronto

Frasi relative alla comanda
Al momento di prendere l'ordinazione, il cameriere registra la comanda
Parte delle comande sono espletate dal barista, parte dal pizzaiolo
Se viene ordinato ad un cameriere da un cliente un prodotto che non è disponibile, questo non potrà essere aggiunto alla comanda

Frasi relative ai tavoli
In tale pizzeria è di interesse tenere traccia dei tavoli disponibili ed occupati
All'ingresso di un gruppo di commensali, il manager li riceve … assegnando un tavolo disponibile in grado di ospitarli tutti
Un cameriere ha sempre la possibilità di visualizzare quali tavoli a lui associati sono occupati e in quali tavoli le relative comande da lui registrate sono state completamente servite
Il cameriere può visualizzare cosa ... deve servire a quale tavolo
In alcuni giorni … vengono utilizzati un numero differente di tavoli
Il manager può definire … quali tavoli sono utilizzati in quali turni

Frasi relative ai clienti
All'ingresso di un gruppo di commensali, il manager li riceve e registra uno di loro, segnando nome, cognome e numero di commensali, assegnando un tavolo disponibile in grado di ospitarli tutti

Frasi relative ai turni
Il manager può definire quali camerieri lavorano in quali turni e quali tavoli sono utilizzati in quali turni

Frasi relative al menu
Il menu è unico per tutti i turni e definito dal manager
Nel menu è necessario anche prevedere aggiunte per le pizze … con i relativi costi

Frasi relative alle pizze
Quando hanno preparato … una pizza, il cameriere può visualizzare cosa è pronto

Frasi relative alle aggiunte
È necessario anche prevedere aggiunte per le pizze… con i relativi costi

Frasi relative alle bevande
Quando hanno preparato una bevanda ..., il cameriere può visualizzare cosa è pronto

Frasi relative ai prodotti
Il manager ha la possibilità di tenere traccia delle disponibilità dei singoli prodotti. In questo modo, se viene ordinato ad un cameriere da un cliente un prodotto che non è disponibile, questo non potrà essere aggiunto alla comanda

Frasi relative agli scontrini
Il manager ha la possibilità di stampare lo scontrino di una comanda

3. Progettazione concettuale

Costruzione dello schema E-R

Per la costruzione dello schema E-R ho deciso di utilizzare una strategia mista, individuando quindi le componenti che descrivono un frammento elementare della realtà di interesse e definendo uno schema scheletro che le contenga. In questo modo sono in grado di fornire una visione unitaria e generale del progetto.

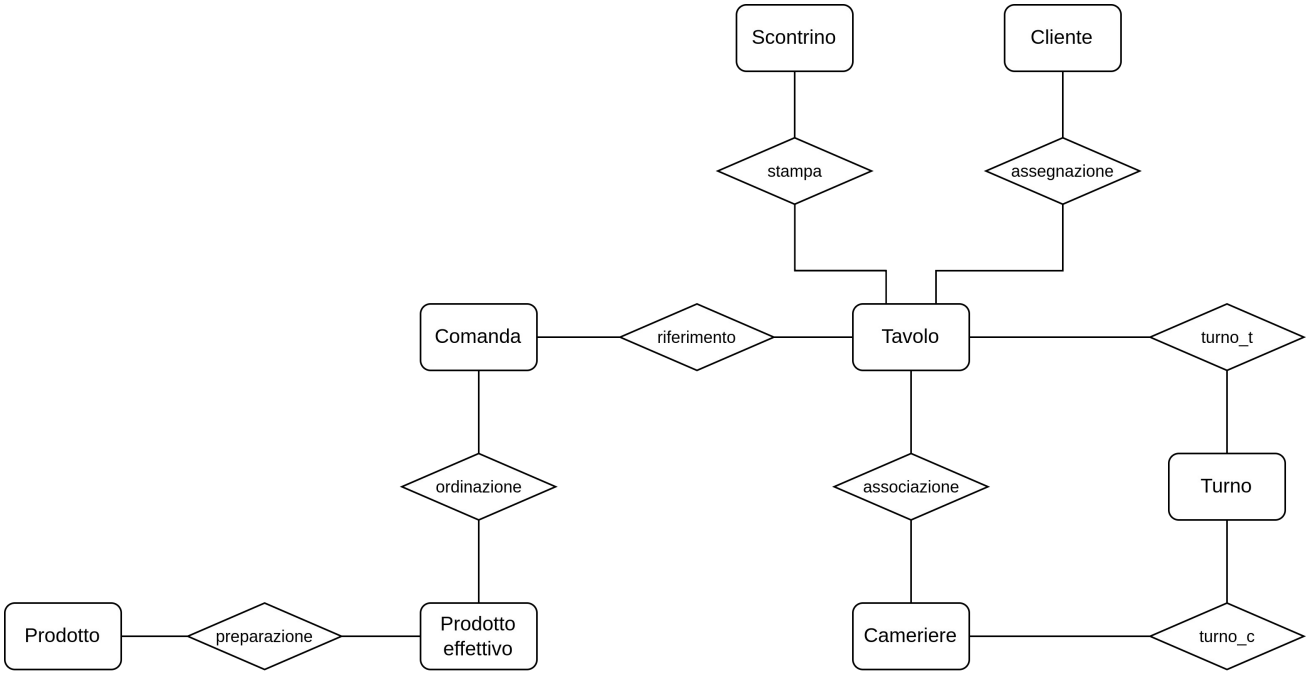

Figura 1

A questo punto sono andata a considerare i concetti principali separatamente, estendendo e raffinando i sottoschemi ricavati.

Inizio analizzando l'entità **Prodotto** ed inserendo gli attributi: "Prezzo" e "Nome", quest'ultimo identifica il prodotto stesso.
È necessario però distinguere tre tipologie di prodotti messi a disposizione dalla pizzeria: le bevande, le pizze e gli ingredienti per poter fare aggiunte alle pizze. Applico quindi una generalizzazione in

cui il padre è l'entità **Prodotto** e le tre entità figlie sono **Ingrediente**, **Pizza** e **Bevanda**.

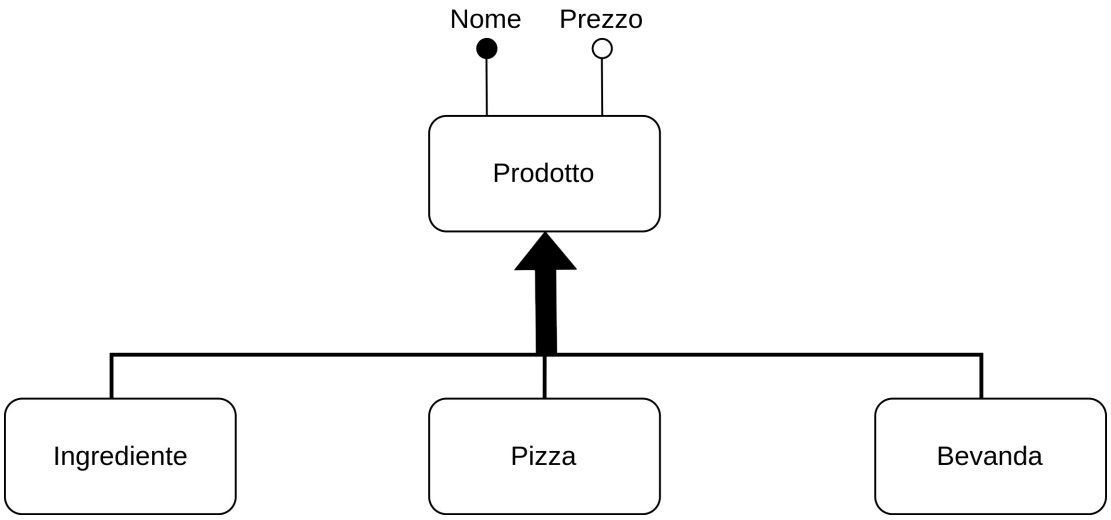

Figura 2

La specifica richiede che si possa tenere traccia delle disponibilità dei singoli prodotti.
Tenendo conto del fatto che all'interno della pizzeria vengano vendute soltanto bevande industriali, non si dovranno indicare gli ingredienti che la compongono e di conseguenza per l'entità **Bevanda** basterà aggiungere un attributo "Quantità" che verrà incrementato di N unità ogni volta che il magazzino verrà rifornito di una quantità pari a N di quella determinata bevanda, verrà invece decrementato di 1 unità ogni volta che essa verrà aggiunta ad una comanda, in modo tale da tener traccia del quantitativo disponibile in magazzino.

Lo stesso vale per l'entità **Ingrediente** poiché gli ingredienti vengono consegnati già pronti alla pizzeria. L'attributo "Quantità" in questo caso verrà decrementato di 1 unità ogni volta che una pizza che contiene tale ingrediente viene ordinata e ogni volta che esso viene usato come aggiunta ad una qualsiasi pizza.

Poiché le pizze non sono prodotti già pronti sarà necessario tener conto degli ingredienti utilizzati per preparare ogni singola pizza e per farlo introduco la relazione ***Condimento***, che sarà interposta tra l'entità **Pizza** e l'entità **Ingrediente**, le cardinalità di tale relazione sono (1, N) per entrambe le entità

perché assumo che ogni pizza debba essere condita con almeno un ingrediente e che all'interno della pizzeria non vengano tenuti ingredienti che non sono utilizzati per condire alcuna pizza.
Inoltre si tenga conto del fatto che ogni ingrediente usato per condire le pizze potrà essere usato anche come aggiunta.

Vado ad aggiungere l'attributo "Disponibilità" all'entità **Pizza**, questo verrà settato a 1 se sono disponibili tutti gli ingredienti utilizzati per condirla (cioè se la loro quantità è diversa da zero), 0 altrimenti.

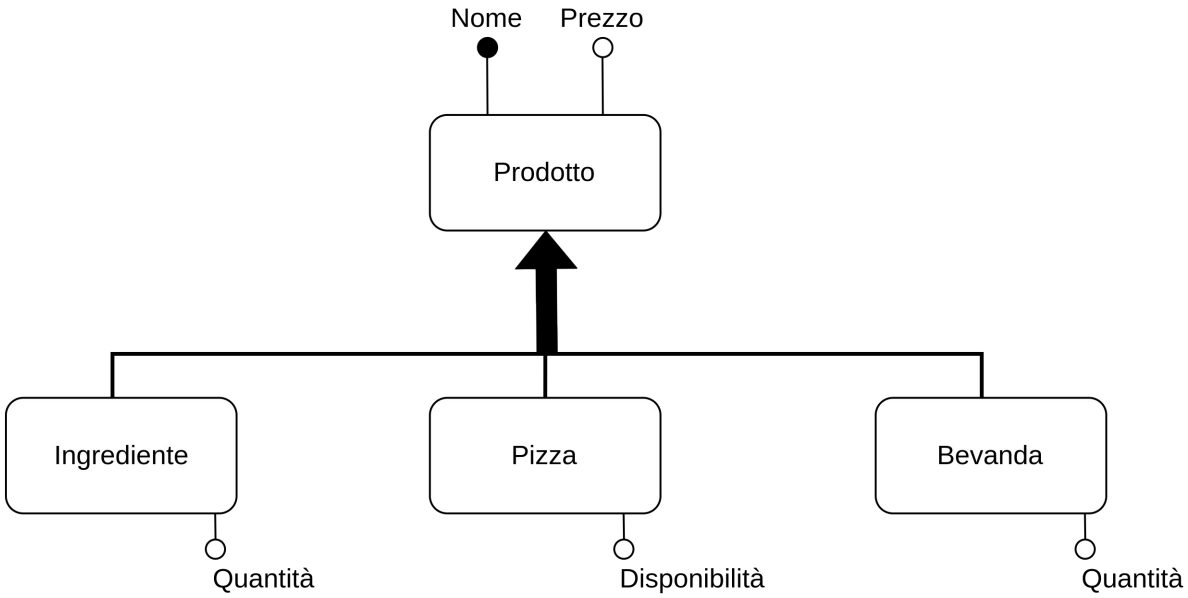

Figura 3

Spostandomi sul concetto di comanda si nota già dallo schema scheletro mostrato in *Figura 1* che diventa necessario introdurre delle ulteriori entità per separare il concetto del prodotto presente nel menu dal prodotto che viene effettivamente ordinato e che quindi dovrà poi essere preparato dal pizzaiolo/barista e servito dal cameriere.

Prima di tutto analizzo l'entità **Comanda** , essa sarà identificata dall'attributo "Numero".
Ho immaginato il concetto di comanda come una sorta di 'archivio' in cui vengono inseriti volta per

volta i prodotti richiesti all'atto dell'ordinazione. Ho considerato quindi che l'ordinazione non venga presa in modo atomico in modo tale da poter permettere ai clienti di inserire nella loro comanda qualsiasi cosa anche in un secondo momento.

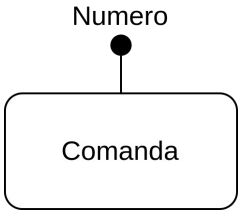

Figura 4

Introduco quindi due relazioni ***Preparazione P*** e ***Preparazione B*** e due entità **Pizza effettiva** e **Bevanda effettiva** e utilizzo il pattern '*instance-of*'.
Uso cardinalità (0, N) per le entità **Bevanda** e **Pizza** perché non necessariamente una pizza o una bevanda presente nel menu dovrà essere ordinata da un cliente e quindi effettivamente preparata.

Vado poi a concentrarmi sul minimondo costituito dalle entità "Bevanda-BevandaEffettiva-Comanda".
Introduco la relazione ***Ordinazione B*** tra l'entità **Bevanda effettiva** e **Comanda**.
L'entità **Bevanda effettiva** avrà un attributo fittizio "Id" per distinguere le varie bevande all'interno della stessa comanda, quindi sarà identificata dalla coppia "Id", "NumeroComanda" (foreign key dell'entità **Comanda**) e "NomeBevanda" (foreign key dell'entità **Bevanda**)

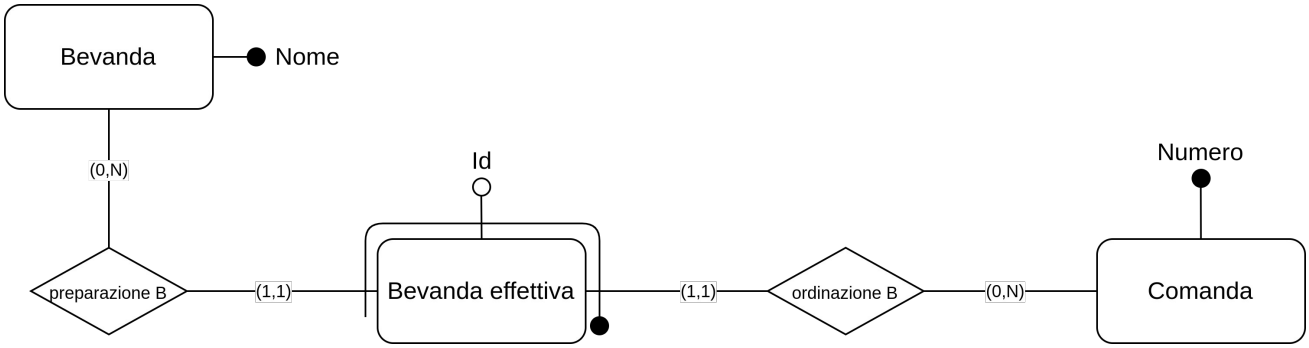

Figura 5

Passando invece al minimondo costituito dalle entità “Pizza-PizzaEffettiva-Comanda” ci saranno alcuni dettagli in più su cui porre l’attenzione.
Il concetto di aggiunta andrà fatto sulla pizza effettiva e non sulla pizza come concetto astratto, utilizzerò quindi una relazione ***Aggiunta*** tra l’entità **Pizza Effettiva** e **Ingrediente**, viene usata l’entità ingrediente e non creata una nuova entità per le aggiunte perché come detto prima tutti gli ingredienti che vengono utilizzati dalla pizzeria come ingredienti per condire le pizze potranno essere anche utilizzati come aggiunte.
Come fatto per le bevande vado ad aggiungere la relazione ***Ordinazione P*** tra l’entità **Pizza effettiva** e l’entità **Comanda** e un attributo fittizio “Id” all’entità **Pizza effettiva** (che permetterà di distinguere le varie pizze all’interno della stessa comanda), anche quest’entità sarà quindi identificata dalla coppia “Id”, “NumeroComanda” (foreign key dell’entità **Comanda**) e “NomePizza” (foreign key dell’entità **Pizza**)

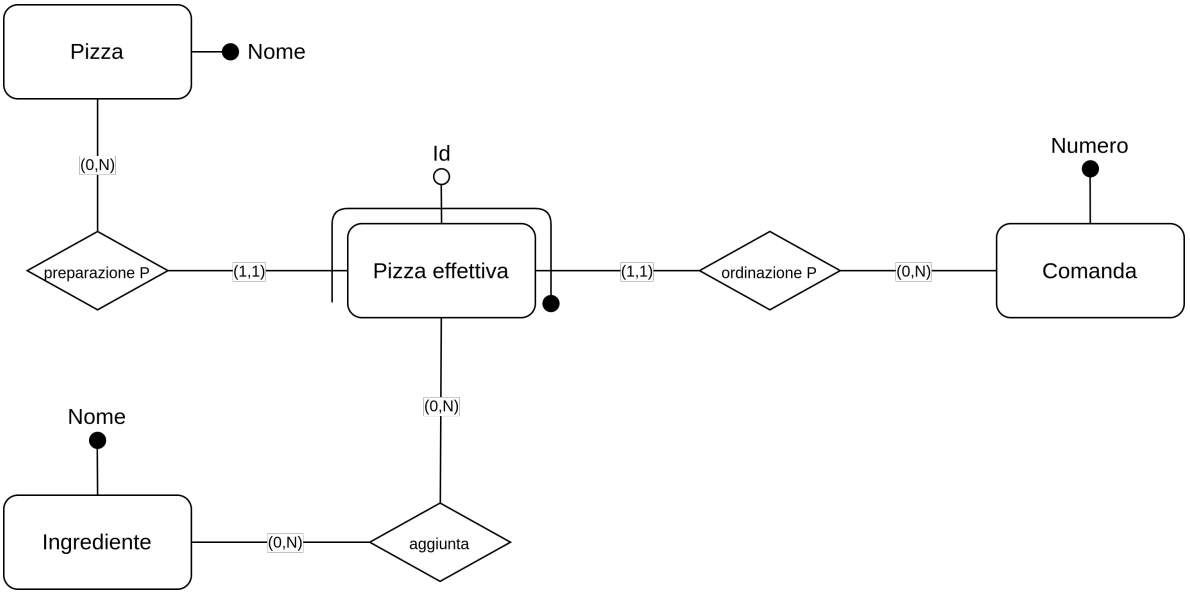

Figura 6

La partecipazione dell’entità **Comanda** sia alla relazione ***Ordinazione P*** che alla relazione ***Ordinazione B*** è opzionale, cioè con cardinalità (0, N), perché immagino ad esempio il caso in cui una comanda è stata appena registrata e non contiene ancora nessuna pizza o bevanda.

Andrò adesso ad analizzare il modo in cui vengono organizzati i tavoli partendo proprio dall'entità **Tavolo** che sarà identificata da un attributo "NumeroTavolo" e, come richiesto dalla specifica, avrà un ulteriore attributo "NumeroPosti" che descrive il numero di posti che esso mette a disposizione. Metto in evidenza il caso in cui un tavolo venga assegnato ad un cliente mediante una generalizzazione parziale che mi permette di rappresentare un caso particolare dell'entità **Tavolo**, cioè l'entità **Tavolo occupato**.

Avrò infatti che una comanda potrà essere registrata solo per un tavolo occupato, introduco quindi la relazione uno a uno ***Riferimento*** tra l'entità **Comanda** e **Tavolo occupato**, quest'ultima entità ha partecipazione opzionale poiché non necessariamente un tavolo che è stato occupato deve avere una comanda che fa riferimento ad esso, immagino ad esempio la situazione in cui un gruppo di commensali si è appena accomodato al tavolo e sta ancora consultando il menu prima di effettuare l'ordinazione. Invece la partecipazione dell'entità **Comanda** è obbligatoria poiché una comanda per essere registrata deve necessariamente far riferimento ad un tavolo.

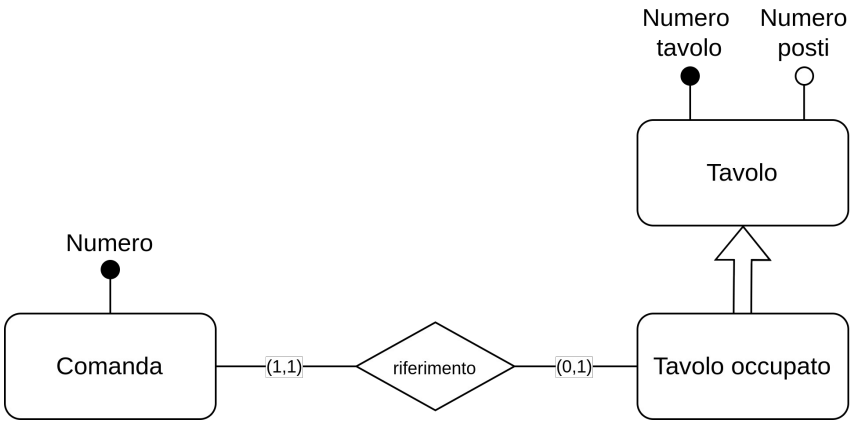

Figura 7

La specifica richiede inoltre che il manager possa stampare gli scontrini, vado quindi ad utilizzare una relazione uno a uno, ***Conto***, tra l'entità **Scontrino** (già presentata nello schema scheletro in *Figura 1*) e l'entità **Tavolo occupato**, questo perché chiaramente se un tavolo non è stato assegnato ad alcun cliente allora non avrò nessuna comanda attiva e non potrò richiedere uno scontrino per tale tavolo.

L'entità **Scontrino** avrà due attributi "Orario" e "Data" che la identificano. Utilizzo la coppia costituita da data e orario come identificatore poiché assumo che non possano essere stampati più scontrini nello stesso istante. Avrò inoltre l'attributo "Prezzo" che sarà dato dalla somma di tutto ciò che era stato inserito in quella determinata comanda e l'attributo "Pagato" che vale 1 se lo scontrino è stato pagato dal cliente, 0 altrimenti.

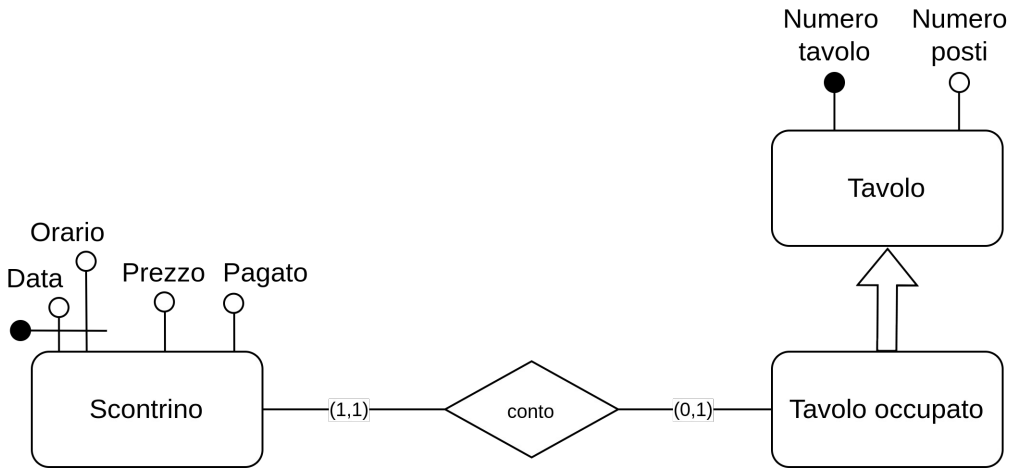

Figura 8

La partecipazione dell'entità **Tavolo occupato** alla relazione ***Conto*** è opzionale poiché posso avere un tavolo per cui non è ancora stato richiesto lo scontrino, mentre quella dell'entità **Scontrino** è obbligatoria perché non può essere richiesta la stampa di uno scontrino per un tavolo che non è occupato da alcun cliente.

Utilizzo inoltre la relazione ***Assegnazione*** per permettere appunto l'assegnazione di un tavolo ad un cliente, tale relazione sarà quindi interposta tra l'entità **Tavolo** e l'entità **Cliente** già mostrata nello schema scheletro.

Il cliente sarà colui che viene preso come 'rappresentanza' dell'intero gruppo di commensali, esso sarà quindi identificato dagli attributi "Nome", "Cognome" e "Tavolo" (identificatore esterno) poiché data la taglia del minimondo assumo che basti tener conto per ciascun cliente del tavolo che gli viene assegnato per risolvere casi di omonimia.

Viene inoltre introdotto l'attributo "NumeroCommensali" e una regola aziendale che impone che ad un cliente possa essere assegnato un tavolo solo se questo mette a disposizione un numero di posti maggiore o uguale al numero di commensali.

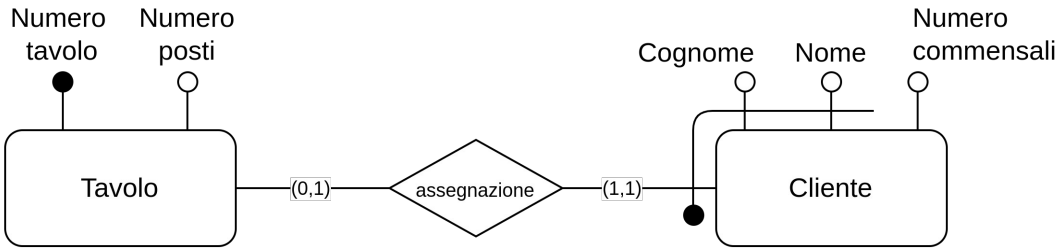

Figura 9

Un altro concetto molto importante da analizzare è quello dell'organizzazione dei turni. Ciò che viene richiesto dalla specifica è che il manager possa definire quali camerieri lavorano in quali turni e quali tavoli sono utilizzati in quali turni. Ho pensato di utilizzare un'entità **Turni** assegnando un nome alle possibili fasce orarie, avrò infatti che i camerieri avranno orari di lavoro 'fissi' con le fasce orarie dei turni definite dal manager (e lo stesso vale per gli orari di utilizzo dei tavoli), ad esempio:

pranzo → 08:00 – 16:00

cena → 16:00 – 24:00

notte → 00:00 – 08:00

L'entità **Turno** sarà quindi identificata dall'attributo "Nome" che mi permetterà di riutilizzare la fascia oraria di tale turno per i camerieri e i tavoli riportando solo il nome che la identifica, e gli altri attributi saranno "OraInizio", "OraFine".

Tramite la relazione ***Turno C*** interposta tra l'entità **Turno** e l'entità **Cameriere** permetto di definire quali camerieri lavorano in quali turni, e allo stesso modo per definire quali tavoli sono usati in quali turni viene introdotta la relazione ***Turno T*** tra l'entità **Tavolo** e l'entità **Turno**.

Entrambe le relazioni **Turno C** e **Turno T** hanno un attributo "Data" per indicare qual è la data effettiva in cui avrò quel determinato cameriere/ tavolo in quel determinato turno.

Infine ho l'entità **Cameriere** identificata dalla coppia di attributi "Nome", "Cognome", questo può bastare sempre perché la taglia del minimondo su cui stiamo lavorando ce lo permette.

Dalle cardinalità si evince che non necessariamente un tavolo o un cameriere debbano essere

associati ad un turno, ma che in un turno devo necessariamente avere almeno un tavolo ed un cameriere.

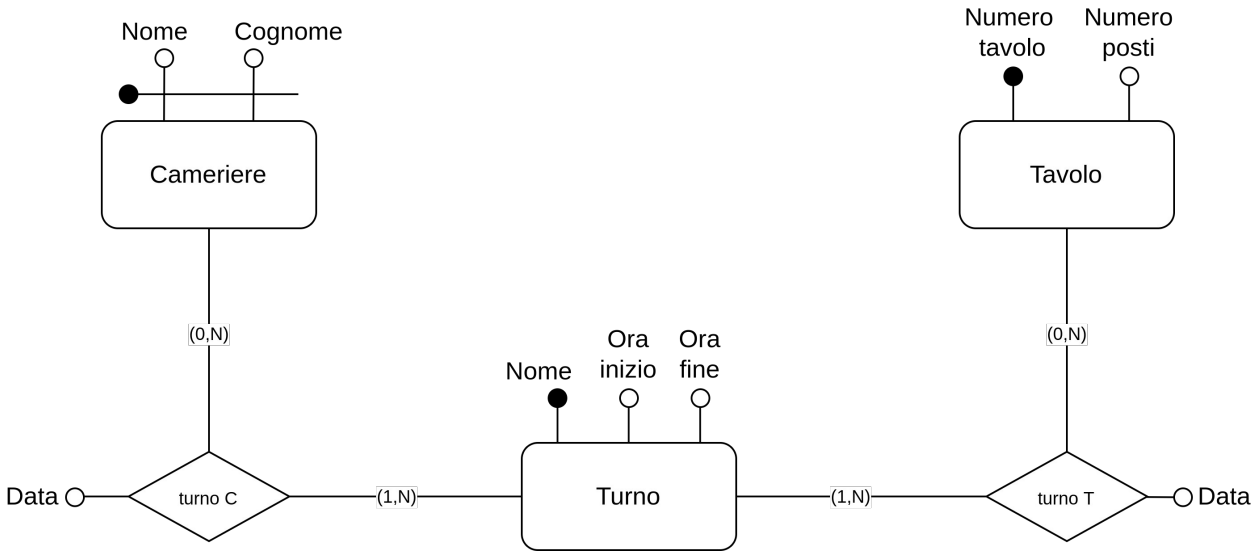

Figura 10

Come ultima cosa vado ad aggiungere una relazione uno a molti, ***Associazione***, tra l'entità **Cameriere** e l'entità **Tavolo**, le cardinalità di tale relazione mostreranno che la partecipazione di entrambe le entità è opzionale poiché non necessariamente un tavolo deve avere un cameriere associato o un cameriere deve essere associato ad un tavolo in un determinato istante, però ad ogni tavolo potrà essere associato al più un cameriere. Sarà però necessario introdurre alcune regole aziendali: una per indicare che per associare un cameriere ad un tavolo devono essere entrambi di turno in quel determinato istante e un'altra per indicare che la registrazione della comanda relativa ad un tavolo potrà essere effettuata solo dal cameriere associato a tale tavolo.

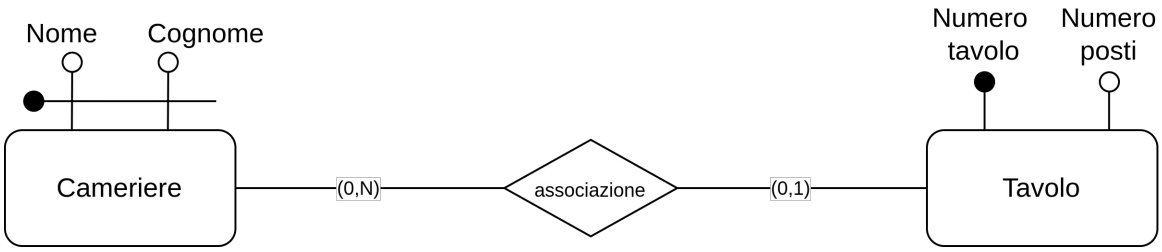

Figura 11

Integrazione finale

Da un'analisi più accurata del concetto di scontrino si deduce la necessità di aggiungere un ulteriore attributo all'entità **Comanda** e una regola aziendale, l'attributo aggiunto è "Servita" e il suo valore sarà pari a 1 se tutto ciò che è stato inserito all'interno della comanda è stato servito, 0 altrimenti.
La regola aziendale consisterà nel permettere la stampa dello scontrino relativo ad un tavolo solo se la comanda che fa riferimento a tale tavolo è stata completamente servita.
Quindi per l'entità **Comanda** otterrò

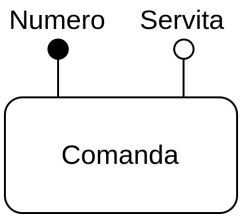

Figura 12

Nello schema finale non utilizzerò l'entità comanda com'era stata rappresentata nelle *Figure 4 – 5 – 6 – 7*, la rappresentazione sarà quindi quella appena introdotta in *Figura 12*.

Inoltre poiché la specifica richiede di permettere al cameriere di visualizzare cosa è pronto (in relazione agli ordini) e sapere cosa deve servire a quale tavolo, ai pizzaiolo e ai baristi di visualizzare cosa devono preparare, aggiungo alle relazioni ***Ordinazione P*** e ***Ordinazione B*** l'attributo "Stato". Tale attributo potrà avere i seguenti valori:

- 0 → ordinata
- 1 → in preparazione
- 2 → pronta
- 3 → servita

Inoltre nel caso delle bevande non potrà essere nullo, mentre nel caso delle pizze si perché in questo modo segnerò una pizza come "ordinata" solo se il cliente non richiede di effettuare aggiunte su di essa o se tutte le aggiunte richieste sono già state applicate.

La rappresentazione usata per le entità **Ingrediente, Pizza** e **Bevanda** sarà quella in *Figura 3* in cui le due entità sono figlie dell'entità **Prodotto** e hanno maggiori dettagli (come "Prezzo", "Disponibilità", "Quantità") rispetto alle rappresentazioni usate nelle altre figure.

Infine ho ritenuto opportuno reificare le due relazioni ***Turno C*** e ***Turno T*** ottenendo così due nuove entità **Turno cameriere** e **Turno tavolo** entrambe in relazione con l'entità **Turno** per tenere traccia dei turni effettivi che verranno definiti dal manager.
L'entità **Turno cameriere** sarà identificata dall'attributo composto "Data" e dalla coppia "Nome" e "Cognome" del cameriere (che sono foreign key dell'entità **Cameriere**), questo perché lo stesso cameriere non può lavorare in più di un turno nella stessa giornata.
Invece l'entità **Turno tavolo** sarà identificata dall'attributo "Data", dal "NumeroTavolo" (foreign key dell'entità **Tavolo**) e dal "Turno" (foreign key dell'entità **Turno**), poiché lo stesso tavolo può essere riutilizzato in più turni della stessa giornata quindi per identificare univocamente tale entità dovrò usare questa tripletta.
Da quest'ultima modifica ottengo il seguente schema parziale che sarà quello utilizzato nello schema finale in sostituzione di quello rappresentato in *Figura 10.*

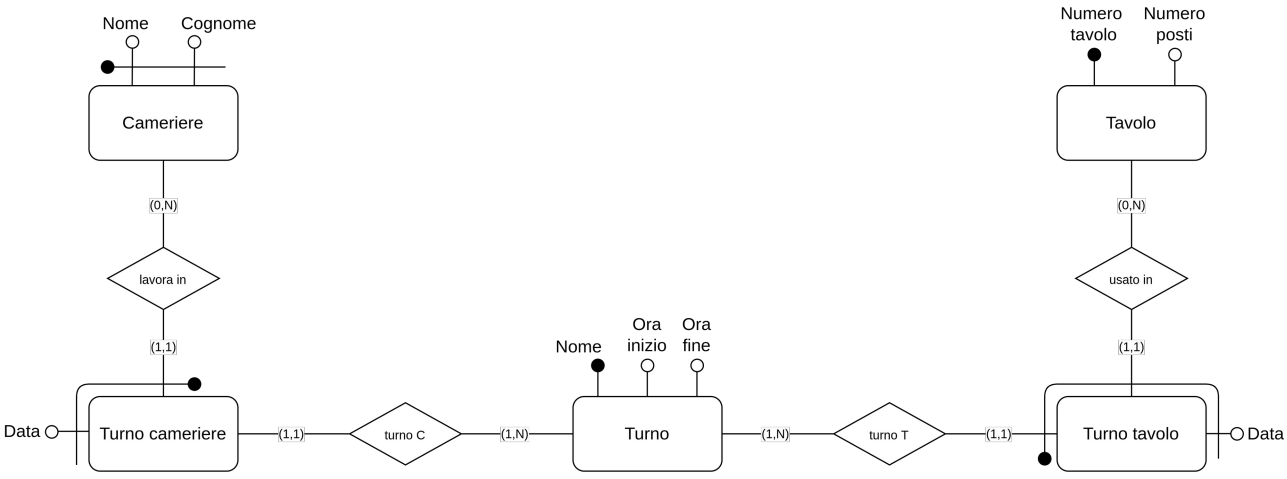

Figura 13

L'unico conflitto sui nomi nasce dall'utilizzo di "turno C" e "turno T" sia prima (*Figura 13*) che dopo la reificazione (*Figura 16*) dello schema mostrato sopra, ovviamente le relazioni non sono le stesse ma per semplicità ho riusato gli stessi nomi.

Lo schema E-R finale è il seguente

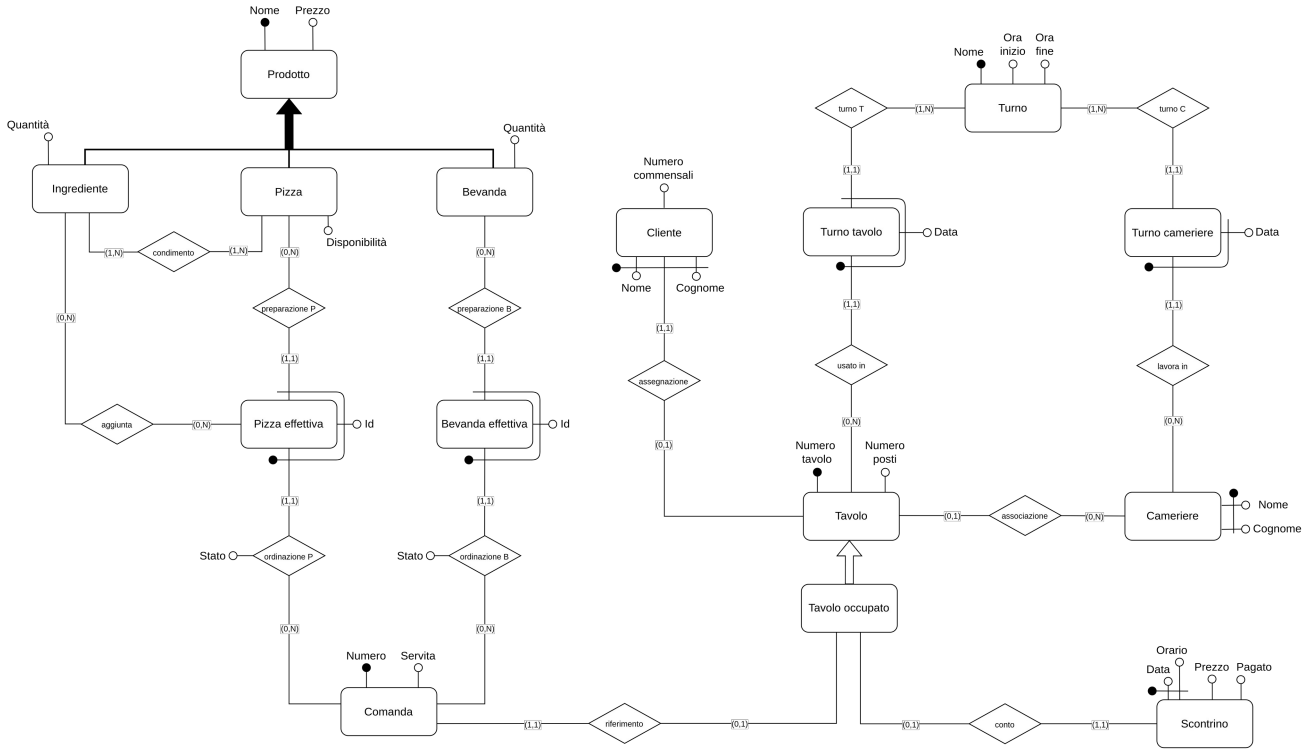

Figura 14

Regole aziendali

- Ad un cliente può essere assegnato un tavolo solo se questo non è occupato, è di turno nell'istante in cui viene richiesta l'assegnazione e mette a disposizione un numero di posti maggiore o uguale al numero di commensali.
- Per associare un cameriere ad un tavolo devono essere entrambi di turno nell'istante in cui viene effettuata l'associazione.
- Per togliere un tavolo ad un cameriere non devono esserci comande attive su di esso. *(Una comanda è considerata inattiva se è stato stampato e pagato lo scontrino relativo al tavolo)*
- Se finisce il turno di un cameriere gli si devono togliere tutti i tavoli associati, se un tavolo ha ancora una comanda attiva si dovrà attendere che essa diventi inattiva per poterlo togliere.
- La registrazione della comanda relativa ad un tavolo (e quindi l'ordinazione delle relative pizze con eventuali aggiute e delle bevande) può essere effettuata solo dal cameriere associato a tale tavolo, egli sarà inoltre l'unico a poter servire ciò che viene ordinato.
- La stampa dello scontrino relativo ad un tavolo può essere effettuata solo se la comanda che fa riferimento a tale tavolo è stata completamente servita.
- Prima di aggiungere un prodotto ad una comanda devo verificare che esso sia disponibile.
- Una pizza è disponibile solo se tutti gli ingredienti usati per condirla hanno quantità diversa da 0
- Tra "OraInizio" e "OraFine" in un Turno devono esserci al massimo 8 ore di differenza.

Dizionario dei dati

Gli identificatori sottolineati sono identificatori esterni

Entità	Descrizione	Attributi	Identificatori
Prodotto	Prodotto presente all'interno del menu della pizzeria (pizze, bevande e ingredienti per le aggiunte)	Nome Prezzo	Nome
Pizza	Entità figlia di 'Prodotto', utilizzata per le pizze che la pizzeria mette a disposizione nel suo menu	Nome Prezzo Disponibilità	Nome
Bevanda	Entità figlia di 'Prodotto', utilizzata per le bevande che la pizzeria mette a disposizione nel suo menu	Nome Prezzo Quantità	Nome
Ingrediente	Entità figlia di 'Prodotto', utilizzata per gli ingredienti messi a disposizione dalla pizzeria nel suo menu, questi vengono usati come condimento per le pizze e/o come aggiunte	Nome Prezzo Quantità	Nome
Pizza effettiva	Pizza che viene effettivamente ordinata dal cliente, inserita quindi nella comanda (con eventuali aggiunte), dovrà essere preparata dal pizzaiolo e servita dal cameriere	Id Stato	Id NumeroComanda NomePizza
Bevanda effettiva	Bevanda che viene effettivamente ordinata dal cliente, inserita quindi nella comanda, dovrà essere preparata dal barista e servita dal cameriere	Id Stato	Id NumeroComanda NomeBevanda
Comanda	Comanda registrata dal cameriere, fa riferimento ad un determinato tavolo e al suo interno vengono inseriti volta per volta i prodotti richiesti dai commensali all'atto dell'ordinazione	Numero Servita	Numero
Cliente	Cliente che viene registrato dal manager, viene preso un membro del gruppo di commensali come 'rappresentanza' dell'intero gruppo	Nome Cognome NumeroCommensali	Nome Cognome Tavolo
Cameriere	Cameriere che lavora all'interno	Nome	Nome

	della pizzeria	Cognome	Cognome
Tavolo	Tavoli che la pizzeria metta a disposizione	NumeroTavolo NumeroPosti	NumeroTavolo
Tavolo occupato	Caso particolare dell'entità 'Tavolo', sono i tavoli che sono stati assegnati dal manager ai clienti	NumeroTavolo NumeroPosti	NumeroTavolo
Scontrino	Scontrino che viene stampato dal manager per un determinato tavolo	Data Orario Prezzo Pagato	Data Orario
Turno	Turni di lavoro con fasce orarie prestabilite	Nome OraInizio OraFine	Nome
Turno tavolo	Fascia oraria in cui verrà utilizzato un determinato tavolo, questo viene definito dal manager e le possibili fasce orari sono quelle descritte da 'Turno'	Data	Data Turno Tavolo
Turno cameriere	Turno di lavoro di un cameriere, definito dal manager	Data	Data Cameriere

4. Progettazione logica

Volume dei dati

Concetto nello schema	Tipo[1]	Volume atteso
Prodotto	E	65
Pizza	E	20
Bevanda	E	10
Ingrediente	E	35
Pizza effettiva	E	75
Bevanda effettiva	E	30
Tavolo	E	30
Tavolo occupato	E	15
Comanda	E	15
Cliente	E	15
Cameriere	E	20
Scontrino	E	36500
Turno	E	3
Turno tavolo	E	300
Turno cameriere	E	75
Condimento	R	80
Preparazione P	R	75
Aggiunta	R	75
Preparazione B	R	30
Ordinazione P	R	75
Ordinazione B	R	30
Assegnazione	R	15
Riferimento	R	15
Conto	R	36500
Associazione	R	15
Turno T	R	300
Usato in	R	300
Turno C	R	75
Lavora in	R	75

Di seguito ho riportato alcuni dettagli sul mondo in cui sono stati stimati i volumi.

[1] Indicare con E le entità, con R le relazioni

- Il volume atteso dell'entità "Pizza effettiva" è 75, infatti una volta che viene effettuata la stampa dello scontrino per un determinato tavolo viene eliminata la comanda che faceva riferimento a tale tavolo e di conseguenza tutte le pizze effettive che appartenevano a tale comanda. Tenendo conto che il volume atteso delle comande è 15 e immaginando che in media ogni comanda contenga 5 pizze effettive avrò 15x5 = 75
- Il volume atteso dell'entità "Bevanda effettiva" è 30, come detto sopra una volta che viene effettuata la stampa dello scontrino per un determinato tavolo viene eliminata la comanda che faceva riferimento a tale tavolo e di conseguenza anche tutte le bevande effettive che appartenevano a tale comanda. Tenendo conto che il volume atteso delle comande è 15 e immaginando che in media ogni comanda contenga 2 bevande effettive avrò 15x2 = 30
- Il volume atteso delle comande e dei clienti è pari a 15 poiché teniamo conto che il volume atteso dei tavoli occupati è 15 e al momento della stampa dello scontrino per un determinato tavolo verrà effettuata la cancellazione della comanda relativa a tale tavolo e del cliente a cui il tavolo era stato assegnato, non siamo interessati a mantenere i dati dei clienti
- Il volume atteso degli scontrini è 36500 poiché immagino di avere circa 100 comande al giorno e poiché gli scontrini verranno eliminati alla fine di ogni anno (in modo tale da permettere al manager di visualizzare le entrate giornaliere e/o mensili) avrò 100x 365 = 36500
- Il volume atteso dell'entità Turno è 3 poiché come già detto in precedenza avrò tre fasce orarie fisse ('Notte', 'Pranzo', 'Cena')
- Il volume atteso dell'entità "Turno tavolo" è 300, infatti ho che il manager ogni domenica definisce i turni per l'intera settimana successiva e ogni giorno vengono cancellati i turni del giorno precedente. Immagino che mediamente i tavoli utilizzati in un turno siano 20, per quanto detto avrò in media i turni di 5 giorni e poiché ho tre turni per giorno ('Notte', 'Pranzo', 'Cena') avrò 20x3x5 = 300
- Il volume atteso dell'entità "Turno cameriere" è 75 , come prima so che il manager ogni domenica definisce i turni per l'intera settimana successiva e ogni giorno vengono cancellati i turni del giorno precedente. Immagino che in media lavorano 5 camerieri per turno, i turni per giorno sono 3 quindi avrò 5x3x5= 75

- Il volume atteso della relazione “Condimento” è 80 poiché abbiamo visto che il volume atteso delle pizze è 20 e supponiamo che mediamente una pizza sia condita con 4 ingredienti
- Il volume atteso della relazione “Aggiunta” è 75, poiché il volume atteso delle comande è 15, immagino che in media ogni comanda contenga 5 pizze effettive e viene effettuata in media 1 aggiunta per pizza, quindi avrò 15x5x1 = 75
- Il volume atteso della relazione “Assegnazione” è 15 poiché immagino di avere mediamente 15 gruppi di commensali all’interno della pizzeria, quindi avrò mediamente 15 assegnazioni di tavoli a clienti. Considero che al momento della stampa dello scontrino per un tavolo venga eliminato il cliente a cui era stato assegnato tale tavolo
- Il volume atteso della relazione “Riferimento” è 15 poiché immagino di avere una comanda per ogni tavolo occupato e il volume atteso dei tavoli occupati è 15
- Il volume atteso della relazione “Associazione” è 15, poiché mediamente i tavoli utilizzati in un turno sono 20, ma suppongo che in media solo 15 tavoli abbiano un cameriere associato

Tavola delle operazioni

Cod.	Descrizione	Frequenza attesa
OP 1	Il manager registra un cliente e assegna ad esso un tavolo	100/giorno
OP 2	Il manager definisce le fasce orarie che saranno usate per i turni	1/anno
OP 3	Il manager definisce quali camerieri lavorano in quali turni	1/settimana
OP 4	Il manager definisce quali tavoli sono utilizzati in quali turni	1/settimana
OP 5	Il manager definisce il menu	1 volta
OP 6	Il manager stampa lo scontrino relativo ad un tavolo	100/giorno
OP 7	Il manager registra il pagamento di uno scontrino relativo ad un tavolo	100/giorno
OP 8	Il manager visualizza le entrate giornaliere	1/giorno
OP 9	Il manager visualizza le entrate mensili	1/mese
OP 10	Il manager associa un cameriere ad un tavolo	150/giorno
OP 11	Il manager toglie un tavolo ad un cameriere	80/giorno
OP 12	Il manager aggiorna la quantità presente in magazzino di un ingrediente	20/settimana
OP 13	Il manager aggiorna la quantità presente in magazzino di una bevanda	5/settimana
OP 14	I camerieri visualizzano quali tavoli a loro associati sono occupati	400/giorno
OP 15	I camerieri visualizzano in quali tavoli le relative comande da loro registrate sono state completamente servite	600/giorno
OP 16	I camerieri registrano una nuova comanda riferita ad un tavolo	100/giorno
OP 17	I camerieri visualizzano cosa è pronto in relazione alle comande prese da loro e a quale tavolo servire	500/giorno

	ciò che è pronto	
OP 18	I camerieri aggiungono una pizza effettiva ad una comanda	500/giorno
OP 19	I camerieri applicano un'aggiunta ad una pizza effettiva	500/giorno
OP 20	I camerieri aggiungono una bevanda effettiva ad una comanda	200/giorno
OP 21	I camerieri servono una pizza	500/giorno
OP 22	I camerieri servono una bevanda	200/giorno
OP 23	I camerieri segnano come ordinata una pizza effettiva	500/giorno
OP 24	I pizzaioli visualizzano le pizze da preparare in ordine di ricezione della comanda	600/giorno
OP 25	I pizzaioli registrano che una pizza è pronta	500/giorno
OP 26	I baristi visualizzano le bevande da preparare in ordine di ricezione della comanda	300/giorno
OP 27	I baristi registrano che una bevanda è pronta	200/giorno

- Per le operazioni 12 e 13 immagino che venga fatto una volta a settimana il rifornimento del magazzino e che in media venga aggiornata la quantità di 20 ingredienti e 5 bevande
- Per l'operazione 24 immagino di avere due pizzaioli per turno che lavorano ed essi visualizzeranno le pizze da preparare più o meno ogni 5 minuti (una giornata è formata da 1440 minuti, 1440/5=288).
- Per l'operazione 26 immagino di avere un solo barista per turno che lavora e visualizzerà le bevande da preparare più o meno ogni 5 minuti.

Costo delle operazioni

Operazione 1: Il manager registra un cliente e assegna ad esso un tavolo, 100/giorno

Suppongo che mediamente ci siano 15 tavoli occupati e tengo conto che questa operazione necessita una verifica sul fatto che il tavolo non sia occupato

Concetto	Costrutto	Accessi	Tipo	Dettagli
Cliente	E	1	S	Registra il cliente
Tavolo occupato	E	15	L	Leggo quali sono i tavoli occupati
Tavolo	E	1	L	Leggo il numero di un tavolo che non era tra quelli occupati, per poterlo assegnare
Assegnazione	R	1	S	Assegna il tavolo al cliente
Tavolo occupato	E	1	S	Segna il tavolo come occupato

Costo: 100 x (3S+16L) = 2200 accessi/giorno

Operazione 2: Il manager definisce le fasce orarie che saranno usate per i turni, 1/anno

Concetto	Costrutto	Accessi	Tipo	Dettagli
Turno	E	3	S	Inserisce 3 turni nel seguente modo : pranzo → 08:00 – 16:00 cena → 16:00 – 24:00 notte → 00:00 – 08:00

Costo: 3S x 1 = 6 accessi/anno

Operazione 3: Il manager definisce quali camerieri lavorano in quali turni, 1/settimana

Suppongo che lavorino in media 5 camerieri per turno, poiché in un giorno ci sono 3 turni avrò che per una settimana dovranno essere definiti 5x3x7 = 105 turni di camerieri

Concetto	Costrutto	Accessi	Tipo	Dettagli
Turno	E	3	L	Leggo le fasce orarie
Cameriere	E	20	L	Leggo i camerieri
Turno C	R	105	S	
Turno cameriere	E	105	S	
Lavora in	R	105	S	

Costo: 315S + 23L = 653 accessi/settimana

Operazione 4: Il manager definisce quali tavoli sono utilizzati in quali turni, 1/settimana

Suppongo che siano utilizzati in media 20 tavoli per turno, poiché in un giorno ci sono 3 turni avrò che per una settimana dovranno essere definiti 20x3x7 = 420

Concetto	Costrutto	Accessi	Tipo	Dettagli
Turno	E	3	L	Leggo le fasce orarie
Tavolo	E	30	L	Leggo i tavoli
Turno T	R	420	S	
Turno tavolo	E	420	S	
Usato in	R	420	S	

Costo: 1260S + 33L= 2553 accessi/settimana

Operazione 5: Il manager definisce il menu, 1 volta

Suppongo che all'interno del menu siano presenti 20 pizze, 10 bevande e 35 ingredienti da poter utilizzare sia come condimenti che per le aggiunte, al momento della definizione del menu dovranno quindi essere inseriti 65 prodotti. Inoltre nella definizione del menu deve essere effettuata anche la definizione dei condimenti di ogni singola pizza, suppongo che mediamente ogni pizza sia condita con 4 ingredienti (4x20 = 80 scritture).

Suppongo inoltre che al momento della definizione del menu siano disponibili tutti gli ingredienti usati per condirli e quindi inizialmente le pizze saranno tutte disponibili

Concetto	Costrutto	Accessi	Tipo	Dettagli
Prodotto	E	65	S	Definisce i prodotti del menu
Ingrediente	E	35	S	Riporto gli ingredienti inseriti nel menu
Pizza	E	20	S	Riporto le pizze inserite nel menu
Bevanda	E	10	S	Riporto le bevande inserite nel menu
Condimento	R	80	S	Definisco i condimenti delle pizze

Costo: (65 + 35 + 20 + 10 + 80)S = 420 accessi

Operazione 6: Il manager stampa lo scontrino relativo ad un tavolo, 100/giorno

La richiesta di stampa di uno scontrino comporta le seguenti operazioni:

- necessità di letture dell'entità 'Tavolo occupato', 'Comanda' e della relazione 'Riferimento' per trovare il numero di comanda associato in quel momento a quel determinato tavolo (se questo non esiste non posso effettuare l'operazione) e verificare che sia stato servito tutto
- necessità di letture delle entità 'Pizza effettiva', 'Bevanda effettiva', 'Prodotto' e delle relazioni 'Ordinazione P', 'Ordinazione B', 'Preparazione P', 'Preparazione B', 'Aggiunta' per determinare il prezzo totale da inserire nello scontrino

Concetto	Costrutto	Accessi	Tipo	Dettagli
Tavolo occupato	E	1	L	Verifico che il tavolo per cui è richiesto lo scontrino esista e sia occupato
Riferimento	R	1	L	Verifico che esista una comanda che riferisce al tavolo richiesto e leggo il numero di comanda
Comanda	E	1	L	Leggo l'attributo 'Servita' per verificare che sia stato servito tutto
Ordinazione B	R	2	L	Ho stimato 2 bevande per comanda
Ordinazione P	R	5	L	Ho stimato 5 pizze per comanda
Bevanda effettiva	E	2	L	
Pizza effettiva	E	5	L	
Aggiunta	R	5	L	Ho 5 pizze per comanda, ciascuna con 1 aggiunta. Vado a leggere quali sono gli ingredienti usati per le aggiunte
Prodotto	E	8	L	Per trovare il prezzo dei singoli prodotti che compongono la comanda e poi sommarli per ottenere il prezzo totale da usare per lo scontrino
Conto	R	1	S	
Scontrino	E	1	S	Inserisco lo scontrino
Scontrino	E	1	L	Leggo gli attributi 'Prezzo' e 'DataOra'

Costo: 100 x (2S + 31L) = 3500 accessi/giorno

Operazione 7: Il manager registra il pagamento di uno scontrino relativo ad un tavolo, 100/giorno

Devo considerare che il pagamento di uno scontrino relativo ad un determinato tavolo comporta altre operazioni:

- eliminazione della comanda che fa riferimento al tavolo per il quale è stato effettuato il pagamento dello scontrino con conseguente eliminazione di tutte le pizze effettive (con le eventuali aggiunte) e le bevande effettive di tale comanda
- eliminazione del cliente a cui era stato assegnato il tavolo per il quale è stato effettuato il pagamento dello scontrino
- eliminazione dell'istanza di 'Tavolo occupato' in cui l'attributo 'NumeroTavolo' è uguale al numero del tavolo per il quale è stato effettuato il pagamento dello scontrino
- lettura dei turni, dei turni dei camerieri, lettura ed eventuale eliminazione dell'associazione cameriere-tavolo

Concetto	Costrutto	Accessi	Tipo	Dettagli
Conto	R	1	L	In base al tavolo su cui è richiesto il pagamento leggo lo scontrino da andare a segnare come pagato
Riferimento	R	1	L	Leggo il numero di comanda che fa riferimento al tavolo per cui viene richiesto il pagamento dello scontrino
Ordinazione B	R	2	L	Leggo quali sono le bevande effettive appartenenti alla comanda letta prima
Ordinazione P	R	5	L	Leggo quali sono le pizze effettive appartenenti alla comanda letta prima
Bevanda effettiva	E	2	S	Elimino le bevande effettive lette
Aggiunta	R	5	S	Elimino le aggiunte fatte alle pizze effettive lette
Pizza effettiva	E	5	S	Elimino le pizze effettive lette
Preparazione B	R	2	S	Elimino le occorrenze in cui ho le bevande effettive lette
Preparazione P	R	5	S	Elimino le occorrenze in cui ho le pizze effettive lette

Ordinazione B	R	2	S	Elimino le occorrenze in cui il numero di comanda è uguale a quello letto prima
Ordinazione P	R	5	S	Elimino le occorrenze in cui il numero di comanda è uguale a quello letto prima
Comanda	E	1	S	Elimino la comanda letta prima
Assegnazione	R	1	L	Leggo il cliente a cui era stato assegnato il tavolo per cui è stata richiesta la stampa dello scontrino
Cliente	E	1	S	Elimino il cliente a cui era stato assegnato questo tavolo
Tavolo occupato	E	1	S	Elimino questo tavolo da 'Tavolo occupato' poiché dopo la stampa dello scontrino sarà considerato libero
Associazione	R	1	L	Leggo il cameriere associato al tavolo
Turni	E	1	L	Leggo il nome del turno in cui l'ora attuale è compresa tra OraInizio e OraFine
Turni camerieri	E	1	L	
Associazione	R	1	S	Elimino l'associazione se il cameriere non dovesse essere di turno

Costo: 100 x (30S + 13L) = 7300 accessi/giorno

Operazione 8: Il manager visualizza le entrate giornaliere, 1/giorno

Si suppone di stampare mediamente 100 scontrini al giorno, andrò a leggere (e poi a sommare) l'attributo 'Prezzo' di tutte le istanze di 'Scontrino' in cui l'attributo 'Data' corrisponde al giorno richiesto

Concetto	Costrutto	Accessi	Tipo	Dettagli
Scontrino	E	100	L	

Costo: 100 accessi/giorno

Operazione 9: Il manager visualizza le entrate mensili, 1/mese

Si suppone di stampare mediamente 100 scontrini al giorno, in un mese ci sono in media 30 giorni, andrò a leggere (e poi a sommare) l'attributo 'Prezzo' di tutte le istanze di 'Scontrino' in cui l'attributo 'Mese' (facente parte dell'attributo composto 'Data') corrisponde al mese richiesto

Concetto	Costrutto	Accessi	Tipo	Dettagli
Scontrino	E	3000	L	

Costo: 3000 accessi/mese

Operazione 10: Il manager associa un cameriere ad un tavolo, 150/giorno

Questa operazione comporta la necessità di verificare che sia il cameriere che il tavolo siano di turno all'istante di richiesta di associazione.

Concetto	Costrutto	Accessi	Tipo	Dettagli
Tavolo	E	1	L	Leggo il tavolo richiesto
Usato in	R	1	L	
Turno tavolo	E	1	L	Verifico che il tavolo sia utilizzato nella data in cui viene effettuata la richiesta di associazione
Turno T	R	1	L	
Cameriere	E	1	L	Leggo il cameriere richiesto
Lavora in	R	1	L	
Turno cameriere	E	1	L	Verifico che il cameriere sia di turno nella data in cui viene effettuata la richiesta di associazione
Turno C	R	1	L	
Turno	E	2	L	Verifico che l'orario in cui viene richiesta l'associazione sia compreso tra 'OraInizio' e 'OraFine' sia per il turno del cameriere sia per il turno del tavolo
Associazione	R	1	S	Effettuo l'associazione

Costo: 150 x (10L + 1S) = 1800 accessi/giorno

Operazione 11: Il manager toglie un tavolo ad un cameriere, 80/giorno

Per togliere un tavolo ad un cameriere è necessario verificare che tale tavolo non abbia una comanda attiva, ricordo che una comanda è considerata inattiva quando è stato stampato e pagato lo scontrino relativo al tavolo.

Concetto	Costrutto	Accessi	Tipo	Dettagli
Associazione	R	1	L	Leggo in 'Associazione' la tupla in cui il numero del tavolo è uguale a quello richiesto e nome e cognome sono quelli del cameriere richiesto (se il cameriere non è associato al tavolo non potrò togliergli tale tavolo)
Riferimento	E	1	L	Verifico se esiste una comanda riferita a tale tavolo
Associazione	R	1	S	Elimino l'associazione

Costo: 80 x (2L + 1S) = 400 accessi/giorno

Operazione 12: Il manager aggiorna la quantità presente in magazzino di un ingrediente, 20/settimana

Se la quantità di tale ingrediente prima dell'aggiornamento era pari a zero devo andare a settare come disponibili tutte le pizze che contenevano tale ingrediente.

Suppongo che ogni ingrediente venga usato per condire mediamente 3 pizze

Concetto	Costrutto	Accessi	Tipo	Dettagli
Ingrediente	E	1	L	Leggo l'ingrediente
Ingrediente	E	1	S	Vado a modificare l'attributo 'Quantità' di tale ingrediente
Condimento	R	3	L	Controllo quali sono le pizze condite con tale ingrediente
Pizza	E	3	S	Vado a settare l'attributo 'Disponibilità' a 1 per indicare che tali pizze sono disponibili

Costo: 20 x (4S + 4L) = 240 accessi/settimana

Operazione 13: Il manager aggiorna la quantità presente in magazzino di una bevanda, 5/settimana

Concetto	Costrutto	Accessi	Tipo	Dettagli
Bevanda	E	1	L	Leggo la bevanda
Bevanda	E	1	S	Vado a modificare l'attributo 'Quantità' di tale bevanda

Costo: 5 x (1S + 1L) = 15 accessi/settimana

Operazione 14: I camerieri visualizzano quali tavoli a loro associati sono occupati, 400/giorno

Suppongo che mediamente un cameriere sia associato a 3 tavoli

Concetto	Costrutto	Accessi	Tipo	Dettagli
Cameriere	E	1	L	
Associazione	R	3	L	Leggo il numero dei tavoli associati a tale cameriere
Tavolo occupato	E	3	L	Verifico quali di questi sono occupati andando a cercare in 'Tavolo occupato' se sono presenti i numeri di tavolo letti prima

Costo: 400 x 7L = 2800 accessi/giorno

Operazione 15: I camerieri visualizzano in quali tavoli le relative comande da loro registrate sono state completamente servite , 600/giorno

Come detto prima ho che ad ogni cameriere sono associati mediamente 3 tavoli, suppongo inoltre che egli abbia registrato le comande che riferiscono a tutti e 3 i tavoli

Concetto	Costrutto	Accessi	Tipo	Dettagli
Cameriere	E	1	L	
Associazione	R	3	L	Leggo il numero dei tavoli a cui il cameriere è associato
Riferimento	R	3	L	Leggo il numero delle comande che riferiscono ai tavoli letti
Comanda	E	3	L	Leggo l'attributo 'Servita' delle comande con numero di comanda uguale a quello letto (per verificare se è stato servito tutto)

Costo: 600 x 10L = 6000 accessi/giorno

Operazione 16: I camerieri registrano una nuova comanda riferita ad un tavolo, 100/giorno

Devo verificare che il cameriere sia associato a tale tavolo e che il tavolo sia occupato

Concetto	Costrutto	Accessi	Tipo	Dettagli
Cameriere	E	1	L	Leggo il cameriere
Tavolo	E	1	L	Leggo il tavolo
Associazione	R	1	L	Verifico che il cameriere sia associato al tavolo
Tavolo occupato	E	1	L	Verifico che il tavolo sia occupato
Riferimento	R	1	S	
Comanda	E	1	S	Inserisco la nuova comanda

Costo: 100 x (4L + 2S) = 800 accessi/giorno

Operazione 17: I camerieri visualizzano cosa è pronto in relazione alle comande prese da loro e a quale tavolo servire ciò che è pronto, 500/giorno

Supponendo che ogni cameriere sia associato mediamente a 3 tavoli, che si abbiano 2 bevande e 5 pizze per comanda e 3 comande per cameriere

Concetto	Costrutto	Accessi	Tipo	Dettagli
Cameriere	E	1	L	
Associazione	R	3	L	Leggo il numero dei tavoli a cui il cameriere è associato
Riferimento	R	3	L	Leggo il numero delle comande riferite ai tavoli letti
Ordinazione B	R	6	L	Leggo l'attributo 'Stato' per visualizzare se la bevanda è pronta
Ordinazione P	R	15	L	Leggo l'attributo 'Stato' per visualizzare se la pizza è pronta

Costo: 500 x 28L = 14000 accessi/giorno

Operazione 18: I camerieri aggiungono una pizza effettiva ad una comanda, 500/giorno

Ricordo che ho stimato che mediamente una pizza è condita con 4 ingredienti

Concetto	Costrutto	Accessi	Tipo	Dettagli
Comanda	E	1	L	
Pizza	E	1	L	Leggo il 'Nome' della pizza richiesta e l'attributo 'Disponibilità'
Preparazione P	R	1	S	
Ordinazione P	R	1	S	
Pizza effettiva	E	1	S	Aggiungo la pizza alla comanda
Comanda	E	1	S	Se l'attributo 'Servita' era settato a 1 devo portarlo a 0 perché ci sarà questa nuova pizza che dovrà essere servita
Condimento	R	4	L	Leggo gli ingredienti usati per condire la pizza
Ingrediente	E	4	L	Leggo la "Quantità" attuale degli ingredienti
Ingrediente	E	4	S	Decremento di 1 la quantità di ogni ingrediente letta prima

Costo: 500 x (10L + 8S) = 13000 accessi/giorno

Operazione 19: I camerieri applicano un'aggiunta ad una pizza effettiva, 500/giorno

Concetto	Costrutto	Accessi	Tipo	Dettagli
Pizza effettiva	E	1	L	
Ingrediente	E	1	L	Leggo l'attributo 'Quantità' per essere sicura di poter usare l'ingrediente coma aggiunta per la pizza
Aggiunta	R	1	S	Inserisco l'aggiunta
Ingrediente	E	1	S	Decremento di 1 la quantità dell'ingrediente usato come aggiunta

Costo: 500 x (2L + 2S) = 3000 accessi/giorno

Operazione 20: I camerieri aggiungono una bevanda effettiva ad una comanda, 200/giorno

Concetto	Costrutto	Accessi	Tipo	Dettagli
Comanda	E	1	L	
Bevanda	E	1	L	Leggo l'attributo 'Quantità' per essere sicura di poter aggiungere la bevanda alla comanda
Preparazione B	R	1	S	
Ordinazione B	R	1	S	
Bevanda effettiva	E	1	S	Aggiungo la bevanda alla comanda
Comanda	E	1	S	Se l'attributo 'Servita' era settato a 1 devo portarlo a 0
Bevanda	E	1	S	Decremento di 1 la quantità della bevanda aggiunta alla comanda

Costo: 200 x (2L + 5S) = 2400 accessi/giorno

Operazione 21: I camerieri servono una pizza, 500/giorno

Concetto	Costrutto	Accessi	Tipo	Dettagli
Ordinazione P	R	1	L	
Ordinazione P	R	1	S	Modifico l'attributo 'Stato' settandolo a "Servita"

Costo: 500 x (1L + 1S) = 1500 accessi/giorno

Operazione 22: I camerieri servono una bevanda, 200/giorno

Concetto	Costrutto	Accessi	Tipo	Dettagli
Ordinazione B	R	1	L	
Ordinazione B	R	1	S	Modifico l'attributo 'Stato' settandolo a "Servita"

Costo: 200 x (1L + 1S) = 600 accessi/giorno

Operazione 23: I camerieri segnano come ordinata una pizza effettiva, 500/giorno

Concetto	Costrutto	Accessi	Tipo	Dettagli
Ordinazione P	R	1	L	
Ordinazione P	R	1	S	Modifico l'attributo 'Stato' settandolo a "Ordinata"

Costo: 500 x (1L + 1S) = 1500 accessi/giorno

Operazione 24: I pizzaioli visualizzano le pizze da preparare in ordine di ricezione della comanda, 600/giorno

Si supponga che mediamente quando viene effettuata questa operazione si abbiano 10 pizze da preparare con in media 1 aggiunta ciascuna, appartenenti a 2 comande diverse.

Ogni pizza è condita mediamente con 4 ingredienti.

Si deve inoltre tenere conto del fatto che il pizzaiolo per preparare le pizze deve visualizzare gli ingredienti da utilizzare per condirle.

Concetto	Costrutto	Accessi	Tipo	Dettagli
Comanda	E	2	L	
Ordinazione P	R	10	L	Leggo l'attributo 'Stato' e preparo solo le pizze in cui leggo "da preparare" e leggo 'Nome' e 'Id' (posso farlo perché sono chiave di Pizza effettiva)
Aggiunta	R	10	L	Leggo le aggiunte da fare alle pizze effettive con gli Id letti
Condimento	R	40	L	Leggo i condimenti (4 per pizza) delle pizze con i nomi letti

Costo: 600 x 62L = 37200 accessi/giorno

Operazione 25: I pizzaioli registrano che una pizza è pronta, 500/giorno

Concetto	Costrutto	Accessi	Tipo	Dettagli
Ordinazione P	R	1	L	
Ordinazione P	R	1	S	Modifico l'attributo 'Stato' settandolo a "Pronta"

Costo: 500 x (1L + 1S) = 1500 accessi/giorno

Operazione 26: I baristi visualizzano le bevande da preparare in ordine di ricezione della comanda, 300/giorno

Supponendo che mediamente quando viene effettuata questa operazione si abbiano 4 pizze da preparare, appartenenti a 2 comande diverse.

Concetto	Costrutto	Accessi	Tipo	Dettagli
Comanda	E	2	L	
Ordinazione B	R	4	L	Leggo le bevande da preparare e il loro 'Stato', preparo solo le bevande in cui leggo "da preparare"

Costo: 300 x 6L = 1800 accessi/giorno

Operazione 27: I baristi registrano che una bevanda è pronta, 200/giorno

Concetto	Costrutto	Accessi	Tipo	Dettagli
Ordinazione B	R	1	L	
Ordinazione B	R	1	S	Modifico l'attributo 'Stato' settandolo a "Pronta"

Costo: 200 x (1L + 1S) = 600 accessi/giorno

Ristrutturazione dello schema E-R

Analisi delle ridondanze

Dopo aver analizzato la situazione ho deciso di aggiungere all'entità **Comanda** un attributo 'Prezzo', tale attributo causa ridondanza perché è un attributo derivabile da altre entità, infatti verrà aggiornato ogni volta che viene aggiunta una **Pizza effettiva** o una **Bevanda effettiva** alla comanda o un'**Aggiunta** ad una pizza effettiva. Questo verrà fatto sommando all'attributo 'Prezzo' il prezzo della pizza effettiva (con eventuali aggiunte) o della bevanda effettiva che viene inserita nella comanda.

L'introduzione di tale attributo porta un piccolo spreco di memoria, infatti assumendo che per memorizzare il prezzo occorrano 4 byte, dato che il volume atteso di **Comanda** è 15, il dato ridondante richiederà solo 60 byte a livello di occupazione di memoria.

Però diminuisce il numero di accessi per l'OP 6

Operazione 6: Il manager stampa lo scontrino relativo ad un tavolo, 100/giorno

Ricordo che si è assunto di avere in media 2 bevande e 5 pizze con 1 aggiunta ciascuna per comanda

Concetto	Costrutto	Accessi	Tipo	Dettagli
Tavolo occupato	E	1	L	Verifico che il tavolo per cui è richiesto lo scontrino esista e sia occupato
Riferimento	R	1	L	Verifico che esista una comanda che riferisce al tavolo richiesto e leggo il numero di comanda
Comanda	E	1	L	Leggo l'attributo 'Servita' per verificare che sia stato servito tutto e l'attributo 'Prezzo' da usare per lo scontrino
Conto	R	1	S	
Scontrino	E	1	S	Inserisco lo scontrino
Scontrino	E	1	L	Leggo gli attributi 'Prezzo' e 'DataOra'

Costo: 100 x (4L + 2S) = 800 accessi/giorno invece di 3500, circa 2700 accessi in meno

La decisione sull'utilizzo del dato ridondante 'Prezzo' nell'entità **Comanda** porta ovviamente ad effettuare operazioni aggiuntive per mantenere aggiornato tale dato. Infatti ogni volta che andrò ad aggiungere alla comanda una nuova pizza effettiva, un'aggiunta ad una pizza effettiva o una nuova bevanda effettiva dovrò andare a leggere il prezzo e sommarlo al prezzo attuale della comanda. Andiamo quindi ad analizzare coma varia il costo delle OP 18, 19 e 20:

Operazione 18: I camerieri aggiungono una pizza effettiva ad una comanda, 500/giorno

Concetto	Costrutto	Accessi	Tipo	Dettagli
Prodotto	E	1	L	Leggo l'attributo 'Prezzo' della Pizza richiesta
Comanda	E	1	L	
Pizza	E	1	L	Leggo il 'Nome' della pizza richiesta e l'attributo 'Disponibilità' per essere sicura di poter aggiungere la pizza alla comanda
Preparazione P	R	1	S	
Ordinazione P	R	1	S	
Pizza effettiva	E	1	S	Aggiungo la pizza alla comanda
Comanda	E	1	S	Aggiorno gli attributi 'Prezzo' e 'Servita' perché ci sarà questa nuova pizza che farà aumentare il prezzo della comanda e che dovrà essere servita
Condimento	R	4	L	Leggo gli ingredienti usati per condire la pizza
Ingrediente	E	4	L	Leggo la "Quantità" attuale degli ingredienti
Ingrediente	E	4	S	Decremento di 1 la quantità di ogni ingrediente letta prima

Costo: 500 x (11L + 8S) = 13500 accessi/giorno invece di 13000, circa 500 accessi in più

Operazione 19: I camerieri applicano un'aggiunta ad una pizza effettiva, 500/giorno

Concetto	Costrutto	Accessi	Tipo	Dettagli
Prodotto	E	1	L	Leggo l'attributo 'Prezzo' dell'Ingrediente richiesto
Pizza effettiva	E	1	L	
Ingrediente	E	1	L	Leggo l'attributo 'Quantità' per essere sicura di poter usare l'ingrediente coma aggiunta per la pizza
Aggiunta	R	1	S	Inserisco l'aggiunta
Ingrediente	E	1	S	Decremento di 1 la quantità dell'ingrediente usato come aggiunta
Comanda	E	1	S	Aggiorno l'attributo 'Prezzo' della comanda sommando il prezzo dell'aggiunta fatta

Costo: 500 x (3L + 3S) = 4500 accessi/giorno invece di 3000, circa 1500 accessi in più

Operazione 20: I camerieri aggiungono una bevanda effettiva ad una comanda, 200/giorno

Concetto	Costrutto	Accessi	Tipo	Dettagli
Prodotto	E	1	L	Leggo l'attributo 'Prezzo' della Bevanda richiesta
Comanda	E	1	L	
Bevanda	E	1	L	Leggo l'attributo 'Quantità' per essere sicura di poter aggiungere la bevanda alla comanda
Preparazione B	R	1	S	
Ordinazione B	R	1	S	
Bevanda effettiva	E	1	S	Aggiungo la bevanda alla comanda
Bevanda	E	1	S	Decremento di 1 la quantità della bevanda aggiunta alla comanda
Comanda	E	1	S	Aggiorno gli attributi 'Prezzo' e 'Servita'

Costo: 200 x (3L + 5S) = 2600 accessi/giorno invece di 2400, circa 200 in più

Inoltre noto che nell'entità **Comanda** viene già utilizzato un attributo derivabile che causa ridondanza, cioè "Servita". Potrei infatti ottenere tale attributo andando a leggere dalle relazioni ***Ordinazione B*** e ***Ordinazione P*** l'attributo "Stato" e risparmiando circa 15 byte (il volume atteso di Comanda è 15 e per memorizzare l'attributo "Servita" occorre 1 byte).
Questo mi porterebbe ad una diminuzione di circa 1400 accessi per le OP 18 e 20 poiché non dovrei andare ad effettuare la scrittura nell'entità **Comanda** per tenere aggiornato l'attributo "Servita".
Per le OP 6 e 15 invece di effettuare l'accesso in lettura a **Comanda** per leggere l'attributo "Servita" e verificare che sia stato servito tutto, dovrei effettuare circa 5 accessi in lettura alla relazione ***Ordinazione P*** per leggere lo stato delle pizze appartenenti alla comanda e 2 alla relazione ***Ordinazione B*** per leggere lo stato delle bevande appartenenti ad ogni comanda.

Nell'operazione 6 gli accessi aumentano poiché si dovranno leggere quali erano le bevande/pizze effettive appartenenti alla comanda e vedere se il loro stato è segnato come 'Servito'

Concetto	Costrutto	Accessi	Tipo	Dettagli
Tavolo occupato	E	1	L	Verifico che il tavolo per cui è richiesto lo scontrino esista e sia occupato
Riferimento	R	1	L	Verifico che esista una comanda che riferisce al tavolo richiesto e leggo il numero di comanda
Comanda	E	1	L	Leggo l'attributo 'Prezzo' da usare per lo scontrino
Ordinazione B	R	2	L	Leggo lo stato delle bevande effettive appartenenti alla comanda letta prima
Ordinazione P	R	5	L	Leggo lo stato delle pizze effettive appartenenti alla comanda letta prima
Conto	R	1	S	
Scontrino	E	1	S	Inserisco lo scontrino
Scontrino	E	1	L	Leggo gli attributi 'Prezzo' e 'DataOra'

Costo: 100 x (11L + 2S) = 1500 accessi/giorno invece di 800, circa 700 accessi in più

Nell'operazione 15 si ha un aumento degli accessi da eseguire poiché supponendo, come detto prima, che ad ogni cameriere sono associati mediamente 3 tavoli e che egli abbia registrato le comande che riferiscono a tutti e 3 i tavoli avrò:

Concetto	Costrutto	Accessi	Tipo	Dettagli
Cameriere	E	1	L	
Associazione	R	3	L	Leggo il numero dei tavoli a cui il cameriere è associato
Riferimento	R	3	L	Leggo il numero delle comande che riferiscono ai tavoli letti
Ordinazione P	R	15	L	Leggo l'attributo 'Stato' per verificare che siano state servite le pizze appartenenti alle comande lette
Ordinazione B	R	6	L	Leggo l'attributo 'Stato' per verificare che siano state servite le bevande appartenenti alle comande lette

Costo: 600 x 28L = 16800 accessi/giorno invece di 6000, circa 10800 in più

In conclusione ho che nell'entità **Comanda** l'utilizzo dell'attributo ridondante "Prezzo" permette di effettuare circa 500 (2700 – 500 – 1500 – 200) accessi al giorno in meno e porta solo un piccolo spreco di memoria (60 byte), mentre l'eliminazione dell'attributo ridondante "Servita" porta un piccolo risparmio di memoria (15 byte), ma obbliga ad effettuare circa 10100 accessi al giorno in più (10800 + 700 – 1400)

Quindi nello schema ER ristrutturato per l'entità **Comanda** utilizzerò la seguente rappresentazione (non quella in *Figura 12* e *14*)

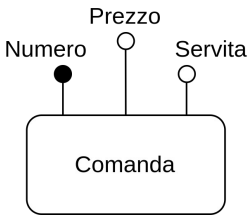

Figura 15

Si è deciso inoltre di mantenere la ridondanza nel caso dell'attributo "Prezzo" dell'entità **Scontrino**, questo perché il sistema è stato progettato in maniera tale che una volta stampato lo scontrino relativo ad un tavolo verrà eliminata la comanda (con tutte le pizze effettive, aggiunte e bevande effettive che appartengono ad essa) che fa riferimento a tale tavolo. Questo, in assenza dell'attributo "Prezzo" nell'entità **Scontrino**, renderebbe impossibile al manager la visualizzazione delle entrate giornaliere e/o mensili.

Passiamo adesso all'attributo "Disponibilità" dell'entità **Pizza**, anche questo è un dato ridondante poiché tale attributo e derivabile dalla lettura dell'attributo "Quantità" degli ingredienti usati per condirla.
Grazie all'eliminazione di tale attributo non ci sarà più la necessità di mantenerlo aggiornato, questo veniva fatto verificando cosa accadeva ad ogni aggiornamento sulla quantità dell'ingrediente:

- se la sua quantità era > 0 e con l'aggiornamento diventava 0 allora era necessario andare a leggere tutte le pizze che lo usavano come condimento e settare la loro disponibilità a 0
- se la sua quantità era 0 e con l'aggiornamento diventava > 0 allora era necessario andare a leggere per ogni pizza che lo usava come condimento se anche tutti gli altri ingredienti usati per condirla avevano quantità > 0 e in caso positivo andare a settare ad 1 la disponibilità della pizza

Vado quindi ad analizzare come cambia il numero degli accessi nelle operazioni coinvolte (OP 12 e OP 18)

Operazione 12: Il manager aggiorna la quantità presente in magazzino di un ingrediente

Concetto	Costrutto	Accessi	Tipo	Dettagli
Ingrediente	E	1	L	Leggo l'ingrediente
Ingrediente	E	1	S	Vado a modificare l'attributo 'Quantità' di tale ingrediente

Costo: 20 x (1S + 1L) =60 accessi/settimana invece di 240, circa 180 accessi in meno alla settimana

Operazione 18: I camerieri aggiungono una pizza effettiva ad una comanda

Ricordo che ho stimato che mediamente una pizza è condita con 4 ingredienti

Concetto	Costrutto	Accessi	Tipo	Dettagli
Comanda	E	1	L	
Pizza	E	1	L	Leggo il 'Nome' della pizza richiesta
Preparazione P	R	1	S	
Ordinazione P	R	1	S	
Pizza effettiva	E	1	S	Aggiungo la pizza alla comanda
Comanda	E	1	S	Se l'attributo 'Servita' era settato a 1 devo portarlo a 0 perché ci sarà questa nuova pizza che dovrà essere servita
Condimento	R	4	L	Leggo gli ingredienti usati per condire la pizza
Ingrediente	E	4	L	Leggo la "Quantità" attuale degli ingredienti
Ingrediente	E	4	S	Decremento di 1 la quantità di ogni ingrediente letta prima

Costo: 500 x (10L + 8S) = 13000 accessi/giorno, il numero di accessi rimane uguale

Quindi l'elliminazione di "Disponibilità" porta, anche se di poco, ad una diminuzione degli accessi (180 in meno a settimana) e ad un risparmio di memoria che prima veniva utilizzata per mantenere tale attributo.

Eliminazione delle generalizzazioni

Decido di eliminare la generalizzazione nel caso delle entità **Tavolo** e **Tavolo occupato** mediante l'accorpamento dell'entità figlia (Tavolo occupato) nell'entità padre (tavolo) aggiungendo all'entità **Tavolo** l'attributo 'Disponibilità' che sarà settato a 0 se il tavolo è già stato assegnato ad un cliente, 1 altrimenti.

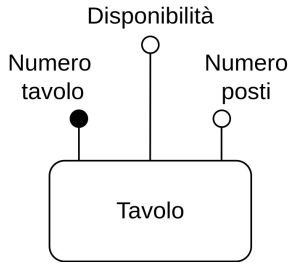

Figura 16

Questa scelta mi porta ad avere un numero di accessi minore per eseguire OP 1

Operazione 1: Il manager registra un cliente e assegna ad esso un tavolo, 100/giorno

Supponendo che mediamente ci siano 15 tavoli occupati

Concetto	Costrutto	Accessi	Tipo	Dettagli
Cliente	E	1	S	Registra il cliente
Tavolo	E	1	L	Scelgo un tavolo qualsiasi leggendo l'attributo 'Disponibilità' che deve essere pari a 1
Assegnazione	R	1	S	Assegna il tavolo al cliente
Tavolo	E	1	S	Setto l'attributo 'Disponibilità' a 0

Costo: 100 x (3S+1L) = 700 accessi/giorno invece di 2200

Nell'OP 6 e invece di leggere dall'entità **Tavolo occupato** leggerò dall'entità **Tavolo** e invece di scrivere su **Tavolo occupato** per eliminarlo andrò a scrivere su **Tavolo** per andare a settare l'attributo 'Disponibilità' a 1, quest'ultima cosa viene fatta anche per l'OP 7.
Quindi il numero di accessi per OP 6 e 7 rimarrà uguale.

Nell'OP 14 invece di leggere dall'entità **Tavolo occupato**, per visualizzare quali tavoli sono occupati, leggerò l'attributo 'Disponibilità' dell'entità **Tavolo** (il numero di accessi rimarrà uguale)

Operazione 14: I camerieri visualizzano quali tavoli a loro associati sono occupati, 400/giorno

Nell'OP 16 non sarà più necessario leggere dall'entità **Tavolo occupato**, ma basterà leggere l'attributo 'Disponibilità' dall'entità **Tavolo** (il numero di accessi diminuisce anche se di poco)

Operazione 16: I camerieri registrano una nuova comanda riferita ad un tavolo, 100/giorno

Devo verificare che il cameriere sia associato a tale tavolo e che il tavolo sia occupato

Concetto	Costrutto	Accessi	Tipo	Dettagli
Cameriere	E	1	L	Leggo il cameriere
Tavolo	E	1	L	Leggo il tavolo e verifico che sia occupato leggendo l'attributo 'Disponibilità'
Associazione	R	1	L	Verifico che il cameriere sia associato al tavolo
Riferimento	R	1	S	
Comanda	E	1	S	Inserisco la nuova comanda

Costo: 100 x (3L + 2S) = 700 accessi/giorno invece di 800

Dovrà essere introdotta una regola aziendale poiché una comanda che fa riferimento ad un tavolo potrà essere registrata solo se tale tavolo è occupato.

Per la generalizzazione che coinvolge le entità **Prodotto**, **Ingrediente**, **Pizza**, **Bevanda** in cui la prima è padre e le altre tre sono figlie *(Figura 3)*, decido di eliminare la generalizzazione mediante l'accorpamento dell'entità padre nelle entità figlie, portando quindi tutte le caratteristiche del padre nelle entità figlie ed eliminando l'entità padre.

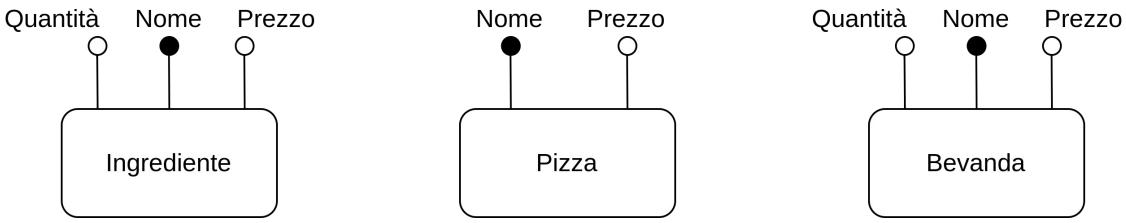

Figura 17

Prendo questa decisione perché la generalizzazione è totale e gli accessi alle entità figlie sono distinti, cioè le operazioni viste sono fatte separatamente su Ingrediente, Pizza o Bevanda.
Questa modifica porta una diminuzione a livello di accessi per l'OP 5

Operazione 5: Il manager definisce il menu, 1 volta

Supponendo come detto prima che all'interno del menu siano presenti 20 pizze, 10 bevande e 35 ingredienti da poter utilizzare sia come condimenti che per le aggiunte e supponendo inoltre che mediamente ogni pizza sia condita con 4 ingredienti.

Concetto	Costrutto	Accessi	Tipo	Dettagli
Ingrediente	E	35	S	Riporto gli ingredienti inseriti nel menu
Pizza	E	20	S	Riporto le pizze inserite nel menu
Bevanda	E	10	S	Riporto le bevande inserite nel menu
Condimento	R	80	S	Definisco i condimenti delle pizze

Costo: (35 + 20 + 10 + 80)S = 290 accessi invece di 420

Inoltre si ha una diminuzione di accessi anche per le OP 18, 19 e 20, infatti per visualizzare il prezzo di un ingrediente, di una pizza o di una bevanda non sarà più necessario accedere in lettura all'entità **Prodotto**, ma basterà accedere rispettivamente alle entità **Ingrediente, Pizza, Bevanda**

Scelta degli identificatori primari

Per le entità **Pizza effettiva** e **Bevanda effettiva** ho scelto di utilizzare come identificatore primario soltanto la coppia “Id”, “NumeroComanda” (foreign key dell’entità **Comanda**) poiché questi attributi sono sufficienti ad identificare univocamente una pizza o una bevanda all’interno di una specifica comanda.

Per il resto gli identificatori primari sono quelli già presentati nello schema ER

Terminata la fase di ristrutturazione otteniamo il seguente schema ER

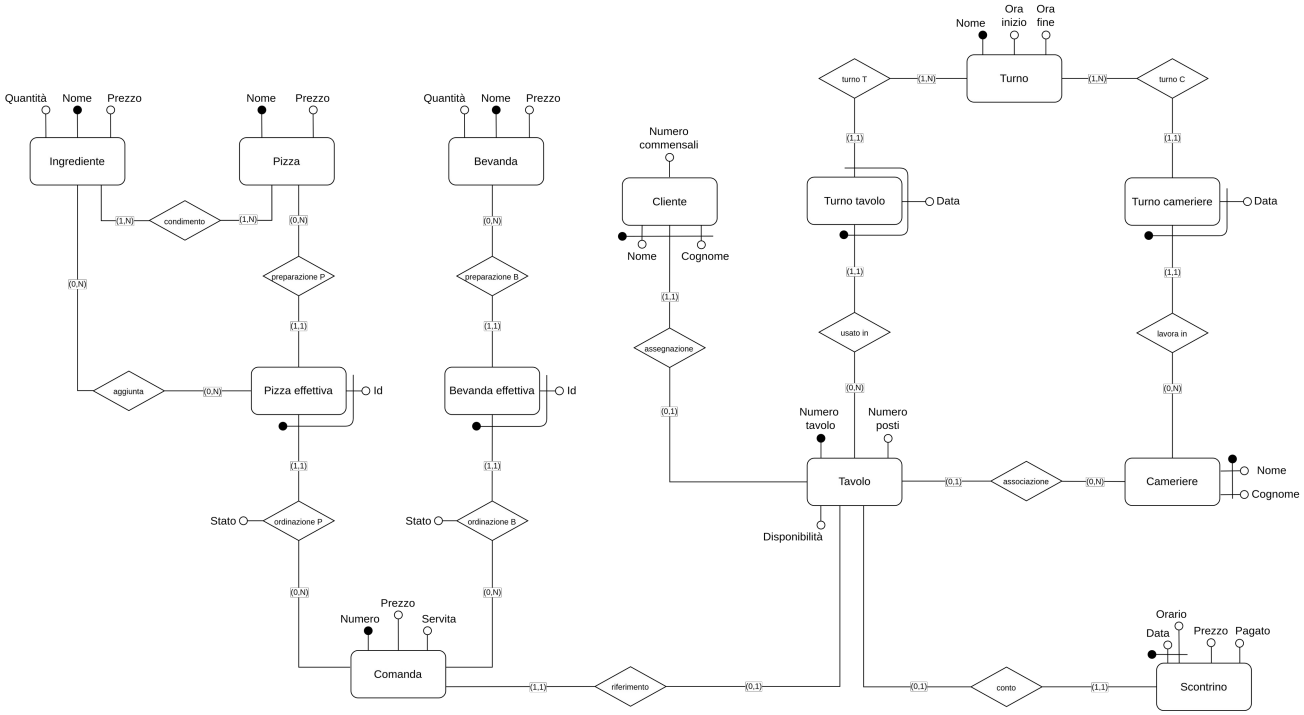

Figura 18

Trasformazione di attributi e identificatori

- L’attributo “Tavolo” in COMANDA, SCONTRINO**,** CLIENTE e TURNO_TAVOLO corrisponde all’attributo “NumeroTavolo” dell’entità **Tavolo**
- L’attributo “Comanda” in PIZZA_EFFETTIVA, BEVANDA_EFFETTIVA e AGGIUNTA corrisponde all’attributo “Numero” dell’entità **Comanda**
- L’attributo “NomePizza” in PIZZA_EFFETTIVA corrisponde all’attributo “Nome” dell’entità **Pizza**
- L’attributo “NomeBevanda” in BEVANDA_EFFETTIVA corrisponde all’attributo “Nome” dell’entità **Bevanda**
- L’attributo “Turno” in TURNO_TAVOLO e TURNO_CAMERIERE corrisponde all’attributo “Nome” dell’entità **Turno**
- Gli attributi “NomeC”, “CognomeC” in TAVOLO corrispondono rispettivamente all’attributo “Nome” e “Cognome” dell’entità **Cameriere**
- Gli attributi “Pizza” e “Ingrediente” in CONDIMENTO corrispondono rispettivamente all’attributo “Nome” dell’entità **Pizza** e all’attributo “Nome” dell’entità **Ingrediente**
- Gli attributo “IdPizza” e “Ingrediente” in AGGIUNTA corrispondono rispettivamente all’attributo “Id” dell’entità **Pizza effettiva** e all’attributo “Nome” dell’entità **Ingrediente**

Traduzione di entità e associazioni

INGREDIENTE (**Nome**, Quantità, Prezzo)

PIZZA (**Nome**, Prezzo)

BEVANDA (**Nome**, Quantità, Prezzo)

TURNO (**Nome**, OraInizio, OraFine)

CAMERIERE (**Nome, Cognome**)

COMANDA (**Numero**, Tavolo, Prezzo, Servita)

SCONTRINO (**Data, Orario**, Prezzo, Pagato, Tavolo)

TAVOLO (**NumeroTavolo**, NumeroPosti, Disponibilità, NomeC*, CognomeC*)

PIZZA_EFFETTIVA (**Id, Comanda**, NomePizza, Stato)

BEVANDA_EFFETTIVA (**Id, Comanda**, NomeBevanda, Stato)

CLIENTE (**Nome, Cognome, Tavolo**, NumeroCommensali)

TURNO_TAVOLO (**Tavolo, Turno, Data**)

TURNO_CAMERIERE (**NomeC, CognomeC, Data**, Turno)

CONDIMENTO (**Pizza, Ingrediente**)

AGGIUNTA (**IdPizza, Comanda, Ingrediente**)

VINCOLI:

COMANDA (Tavolo) ⊆ TAVOLO (NumeroTavolo)

SCONTRINO (Tavolo) ⊆ TAVOLO (NumeroTavolo)

TAVOLO (NomeC, CognomeC) ⊆ CAMERIERE (Nome, Cognome)

PIZZA_EFFETTIVA (Comanda) ⊆ COMANDA (Numero)

PIZZA_EFFETTIVA (NomePizza) ⊆ PIZZA (Nome)

BEVANDA_EFFETTIVA (Comanda) ⊆ COMANDA (Numero)

BEVANDA_EFFETTIVA (NomeBevanda) ⊆ BEVANDA (Nome)

CLIENTE (Tavolo) ⊆ TAVOLO (NumeroTavolo)

TURNO_TAVOLO (Tavolo) ⊆ TAVOLO (NumeroTavolo)

TURNO_TAVOLO (Turno) ⊆ TURNO (Nome)

TURNO_CAMERIERE (NomeC, CognomeC) ⊆ CAMERIERE (Nome, Cognome)

TURNO_ CAMERIERE (Turno) ⊆ TURNO (Nome)

CONDIMENTO (Pizza) ⊆ PIZZA (Nome)

CONDIMENTO (Ingrediente) ⊆ INGREDIENTE (Nome)

AGGIUNTA (IdPizza, Comanda) ⊆ PIZZA_EFFETTIVA (Id, Comanda)

AGGIUNTA (Ingrediente) ⊆ INGREDIENTE (Nome)

Normalizzazione del modello relazionale

1ª FORMA NORMALE

La prima forma normale è rispettata poiché è sempre presente una chiave primaria e quindi non quindi non esistono tuple duplicate, non sono presenti gruppi di attributi che si ripetono e tutte le colonne sono indivisibili, cioè non vengono utilizzati attributi composti.

2ª FORMA NORMALE

Anche la seconda forma normale è rispettata, infatti non sono presenti dipendenze parziali, tutti gli attributi che non appartengono alle chiavi minime sono determinati dall'intera chiave.

3ª FORMA NORMALE

Infine dimostro che la terza forma normale è rispettata poiché non si verifica mai che sia presente un campo che non dipende dalla chiave, cioè non sono presenti dipendenze transitive.

5. Progettazione fisica

Utenti e privilegi

All'interno dell'applicazione sono stati previsti 4 tipi di utente:

- Manager
- Cameriere
- Pizzaiolo
- Barista

Un ruolo è un insieme di GRANT, permette cioè di fornire all'utente del database la capacità di effettuare operazioni su una determinata risorsa.

Per identificare le persone fisiche (ciascuna appartenente ad una classe di utenti e caratterizzata quindi da uno specifico ruolo) che potranno accedere al sistema verranno utilizzati un nome utente ed una password, in particolare la registrazione utilizzando un nome utente della forma *"nome_cognome"* in modo tale da poter effettuare le verifiche sull'associazione cameriere-tavolo.

Viene introdotto un utene di database fittizio Login, questo permette di instaurare una connessione client-server prima ancora di sapere chi è l'utente che tenterà di connettersi alla base di dati. Tale utente avrà come unico privilegio l'esecuzione della procedura "*login*", in questo modo garantisco che anche se ho un utente che non appartiene a nessun ruolo e non è autenticato (cioè non è presente all'interno della tabella "Utenti") potrà soltanto eseguire la procedura di login.

In base al ruolo di chi effettua il login verrà cambiato l'utente di database con cui si è connessi portando in caso di successo ad un'elevazione di privilegi (rispetto all'utente login).

I privilegi previsti per ogni utente sono forniti tramite i ruoli soltanto sull'esecuzione delle procedure e non direttamente sulle tabelle, in modo tale da permettere loro di apportare modifiche alle tabelle solo tramite le stored procedure messe a loro disposizione.

Di seguito sono riportate, per ogni tipologia di utente, le procedure su cui essi hanno come privilegio EXECUTE.

Login

- login

Manager

- crea_turno
- crea_turno_cameriere
- crea_turno_tavolo
- associa_cameriere_tavolo
- togli_cameriere_tavolo
- registra_cliente
- inserisci_pizza_menu
- inserisci_ingrediente_menu
- inserisci_condimento_pizza
- inserisci_bevanda_menu
- aggiorna_quantita_ingrediente
- aggiorna_quantita_bevanda
- stampa_scontrino
- registra_pagamento_scontrino
- visualizza_entrata_giornaliera
- visualizza_entrata_mensile
- inserisci_tavolo
- inserisci_cameriere
- cancella_turno_cameriere

Cameriere

- visualizza_tavoli_occupati
- visualizza_tavoli_comande_servite
- registra_comanda
- visualizza_cosa_e_dove_servire
- ordina_pizza
- ordina_bevanda
- segna_pizza_servita
- segna_bevanda_servita

Pizzaiolo

- visualizza_pizze_da_preparare
- segna_pizza_pronta

Barista

- visualizza_bevande_da_preparare
- segna_bevanda_pronta

Strutture di memorizzazione

Tabella Ingredienti		
Colonna	**Tipo di dato**	**Attributi**[2]
Nome	VARCHAR(45)	PK, NN
Quantità	INT	NN, UN
Prezzo	DECIMAL(3,2)	NN

Tabella Pizze		
Colonna	**Tipo di dato**	**Attributi**
Nome	VARCHAR(45)	PK, NN
Prezzo	DECIMAL(4,2)	NN

Tabella Bevande		
Colonna	**Tipo di dato**	**Attributi**
Nome	VARCHAR(45)	PK, NN
Quantità	INT	NN, UN
Prezzo	DECIMAL(5,2)	NN

[2] PK = primary key, NN = not null, UQ = unique, UN = unsigned, AI = auto increment. È ovviamente possibile specificare più di un attributo per ciascuna colonna.

Tabella Turni		
Colonna	**Tipo di dato**	**Attributi**
Nome	VARCHAR(45)	PK, NN
OraInizio	TIME	NN
OraFine	TIME	NN

Tabella Camerieri		
Colonna	**Tipo di dato**	**Attributi**
Nome	VARCHAR(45)	PK, NN
Cognome	VARCHAR(45)	PK, NN

Tabella Comande		
Colonna	**Tipo di dato**	**Attributi**
Numero	INT	PK, NN, UN, AI
Tavolo	INT	NN, UQ, UN
Prezzo	DECIMAL(6,2)	NN
Servita	TINYINT(1)	NN, UN

Tabella Scontrini		
Colonna	**Tipo di dato**	**Attributi**
DataOra	DATETIME	PK, NN
Prezzo	DECIMAL(6,2)	NN
Pagato	TINYINT(1)	NN, UN
Tavolo	INT	NN, UN

Tabella Tavoli		
Colonna	**Tipo di dato**	**Attributi**
NumeroTavolo	INT	PK, NN, UN
NumeroPosti	INT	NN, UN
Disponibilità	TINYINT(1)	NN, UN
NomeC	VARCHAR(45)	
CognomeC	VARCHAR(45)	

Tabella Pizze_effettive		
Colonna	**Tipo di dato**	**Attributi**
Id	INT	PK, NN, UN
Comanda	INT	PK, NN, UN
NomePizza	VARCHAR(45)	NN
Stato	TINYINT(1)	UN

Tabella Bevande_effettive		
Colonna	**Tipo di dato**	**Attributi**
Id	INT	PK, NN, UN
Comanda	INT	PK, NN, UN
NomeBevanda	VARCHAR(45)	NN
Stato	TINYINT(1)	NN, UN

Tabella Clienti		
Colonna	**Tipo di dato**	**Attributi**
Nome	VARCHAR(45)	PK, NN
Cognome	VARCHAR(45)	PK, NN
Tavolo	INT	PK, NN, UQ, UN
NumeroCommensali	INT	NN, UN

Tabella Turni_Tavoli		
Colonna	**Tipo di dato**	**Attributi**
Tavolo	INT	PK, NN, UN
Turno	VARCHAR(45)	PK, NN
Data	DATE	PK, NN

Tabella Turni_Camerieri		
Colonna	**Tipo di dato**	**Attributi**
NomeC	VARCHAR(45)	PK, NN
CognomeC	VARCHAR(45)	PK, NN
Turno	VARCHAR(45)	NN
Data	DATE	PK, NN

Tabella Condimenti		
Colonna	**Tipo di dato**	**Attributi**
Pizza	VARCHAR(45)	PK, NN
Ingrediente	VARCHAR(45)	PK, NN

Tabella Aggiunte		
Colonna	**Tipo di dato**	**Attributi**
IdPizza	INT	PK, NN, UN
Comanda	INT	PK, NN, UN
Ingrediente	VARCHAR(45)	PK, NN

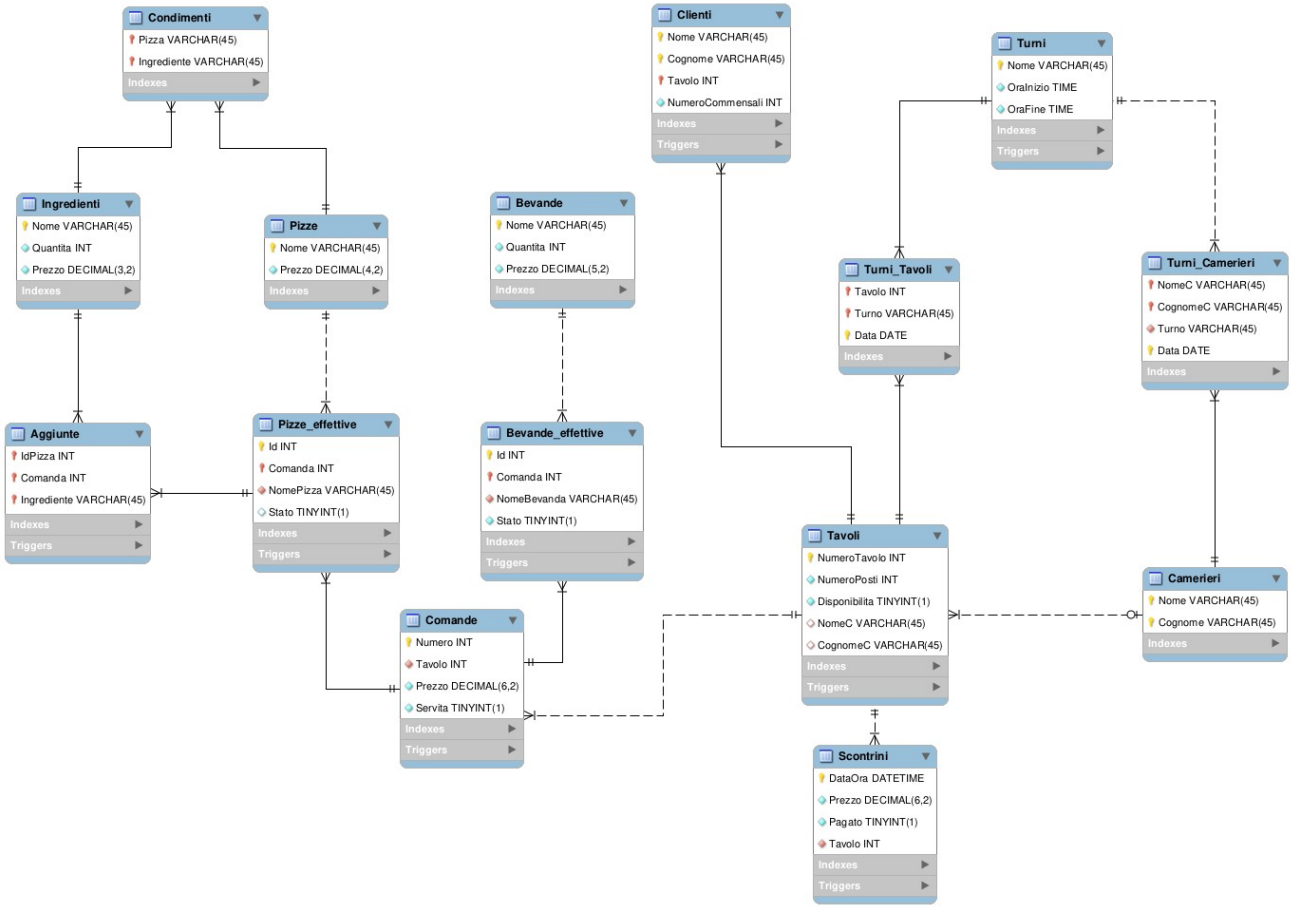

Figura 19

Indici

Sono stati introdotti indici di tipo INDEX per le foreign key, il loro scopo è quello di aumentare la velocità di accesso ai record.
Mentre per le chiavi primarie sono stati introdotti indici di tipo PRIMARY che consentono di identificare univocamente e direttamente ogni singolo record all'interno di una tabella.
Si è deciso inoltre di introdurre degli indici di tipo UNIQUE per 'Tavolo' sia nella tabella **Cliente** che nella tabella **Comanda** poiché in questo modo, dato un tavolo, è possibile garantire che contemporaneamente non vengano assegnati più clienti ad esso e che non vengano registrate più comande relative ad esso.
È stato utilizzato un indice di tipo UNIQUE anche all'interno della tabella **Turni** per gli attributi 'OraInizio' e 'OraFine', in questo modo garantisco che non vengano creati più turni che pur avendo nomi diversi sono caretterizzati dalle stesse fasce orarie.

Tabella Comande	
Indice PRIMARY	**Tipo[3]:**
Numero	PR
Indice Tavolo_UNIQUE	**Tipo:**
Tavolo	UQ

Tabella Bevande	
Indice PRIMARY	**Tipo:**
Nome	PR

Tabella Pizze	
Indice PRIMARY	**Tipo:**
Nome	PR

[3] IDX = index, UQ = unique, FT = full text, PR = primary.

Tabella Ingredienti	
Indice PRIMARY	**Tipo:**
Nome	PR

Tabella Bevande_effettive	
Indice PRIMARY	**Tipo:**
Id, Comanda	PR
Indice fk_BevandeEffettiva_Comanda_idx	**Tipo:**
Comanda	IDX
Indice fk_BevandeEffettiva_Bevanda_idx	**Tipo:**
NomeBevanda	IDX

Tabella Pizze_effettive	
Indice PRIMARY	**Tipo:**
Id, Comanda	PR
Indice fk_PizzeEffettive_Comanda_idx	**Tipo:**
Comanda	IDX
Indice fk_PizzeEffettive_Pizza_idx	**Tipo:**
NomePizza	IDX

Tabella Aggiunte	
Indice PRIMARY	**Tipo:**
IdPizza, Comanda, Ingrediente	PR
Indice fk_Aggiunta_Ingrediente_idx	**Tipo:**
Ingrediente	IDX
Indice fk_Aggiunta_IdPizza_idx	**Tipo:**
IdPizza, Comanda	IDX

Tabella Condimenti	
Indice PRIMARY	**Tipo:**
Pizza, Ingrediente	PR
Indice fk_Condimenti_Ingrediente_idx	**Tipo:**
Ingrediente	IDX
Indice fk_Condimenti_Pizza_idx	**Tipo:**
Pizza	IDX

Tabella Clienti	
Indice PRIMARY	**Tipo:**
Nome, Cognome, Tavolo	PR
Indice Tavolo_UNIQUE	**Tipo:**
Tavolo	UQ

Tabella Tavoli	
Indice PRIMARY	**Tipo:**
NumeroTavolo	PR
Indice fk_Tavoli_Cameriere_idx	**Tipo:**
NomeC, CognomeC	IDX

Tabella Camerieri	
Indice PRIMARY	**Tipo:**
Nome, Cognome	PR

Tabella Turni	
Indice PRIMARY	**Tipo:**
Nome	PR
Indice FasciaOraria_UNIQUE	**Tipo:**
OraInizio, OraFine	UQ

Tabella Turni_Camerieri	
Indice PRIMARY	**Tipo:**
NomeC, CognomeC, Data	PR
Indice fk_TurniC_Turno_idx	**Tipo:**
Turno	IDX
Indice fk_TurniC_Cameriere_idx	**Tipo:**
NomeC, CognomeC	IDX

Tabella Turni_Tavoli	
Indice PRIMARY	**Tipo:**
Tavolo, Turno, Data	PR
Indice fk_TurniT_Turno_idx	**Tipo:**
Turno	IDX
Indice fk_TurniT_Tavolo_idx	**Tipo:**
Tavolo	IDX

Tabella Scontrini	
Indice PRIMARY	**Tipo:**
DataOra	PR
Indice fk_Scontrini_Tavolo_idx	**Tipo:**
Tavolo	IDX

Trigger

verifica_fase_ordinazione

Il seguente trigger viene utilizzato prima di effettuare un inserimento in Aggiunte poiché non potranno essere effettuate aggiunte ad una pizza che non è in fase di ordinazione *(Stato → null).*

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`verifica_fase_ordinazione`
BEFORE INSERT ON `Aggiunte` FOR EACH ROW
BEGIN
	if (SELECT Stato
		FROM Pizze_effettive
		WHERE Id = new.IdPizza
			AND Comanda = new.Comanda) is not null then
		signal sqlstate '45000' set message_text = 'pizza non in fase di ordinazione';
	end if;
END
```

decremento_ingrediente_a

Il seguente trigger va a decrementare di 1 unità la quantità presente in magazzino dell'ingrediente che viene richiesto come aggiunta.
Questo mi permette anche di verificare se l'ingrediente è disponibile perché essendo "Quantita" un intero unsigned esso non potrà mai avere un valore inferiore allo zero, quindi se si prova a decrementare la quantità di un ingrediente che è gia a 0 verrà restituito errore

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`decremento_ingrediente_a`
BEFORE INSERT ON `Aggiunte` FOR EACH ROW
BEGIN
	UPDATE Ingredienti
	SET Quantita = Quantita-1
	WHERE Nome = new.Ingrediente;
END
```

aggiorna_stato_comanda_a

Il seguente trigger viene attivato dopo aver effettuato l'aggiunta alla pizza effettiva e va ad aggiornare il prezzo della comanda sommando al prezzo attuale il prezzo dell'ingredienteusato come aggiunta.

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`aggiorna_stato_comanda_a`
AFTER INSERT ON `Aggiunte` FOR EACH ROW
BEGIN
	UPDATE Comande
	SET Prezzo = Prezzo + (select Prezzo from Ingredienti where Nome = new.Ingrediente)
	WHERE Numero = new.Comanda;
END
```

decremento_bevanda

Come nel caso delle aggiunte, prima di effettuare l'ordinazione di una bevanda (e quindi l'inserimento in *Bevande_effettive*) viene decrementata di uno la quantità della bevanda richiesta.

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`decremento_bevanda`
BEFORE INSERT ON `Bevande_effettive` FOR EACH ROW
BEGIN
	UPDATE Bevande
	SET Quantita = Quantita-1
	WHERE Nome = new.NomeBevanda;
END
```

aggiornamento_stato_comanda_b

Il seguente trigger va ad aggiornare il prezzo della comanda come nel caso introdotto prima per le aggiunte, ma in più quando viene ordinata una bevanda si dovrà andare a verificare se lo stato della comanda per cui viene richiesta quest'ordinazione è 1 (indica che è stato servito tutto ciò che appartiene a tale comanda) in caso lo fosse bisognerà settare questo valore a 0 dato che ci sarà la nuova bevanda appena ordinata da servire.

```
CREATE DEFINER = CURRENT_USER TRIGGER
`pizzeriaDB`.`aggiornamento_stato_comanda_b` AFTER INSERT ON `Bevande_effettive`
FOR EACH ROW
BEGIN
        UPDATE Comande
        SET Prezzo = Prezzo + (select Prezzo from Bevande where Nome = new.NomeBevanda)
        WHERE Numero = new.Comanda;

   if (SELECT Servita
        FROM Comande
        WHERE Numero = new.Comanda) then

        UPDATE Comande
        SET Servita = 0
        WHERE Numero = new.Comanda;

   end if;
END
```

verifica_stato_bevanda

Il seguente trigger viene utilizzato per verificare che:

- quando viene segnata come ‘in preparazione’ (1) la bevanda prima si trovava nello stato ‘ordinata’ (0), questo viene fatto nel momento in cui un barista visualizza le bevande da preparare e prende in carico la preparazione di esse, in modo tale da non permettere che più camerieri visualizzino e preparino più volte la stessa bevanda
- quando viene segnata come ‘pronta’ (2) prima si trovava nello stato ‘in preparazione’ (1), perché una bevanda non può essere preparata se prima non è stata visulizzata e presa in carico da un barista
- quando viene segnata come ‘servita’ (3) prima si trovava nello stato ‘pronta’ (2), questo viene fatto perché chiaramente non può essere servita una bevanda che non è ancora stata preparata dal barista

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`verifica_stato_bevanda`
BEFORE UPDATE ON `Bevande_effettive` FOR EACH ROW
BEGIN
	if new.Stato = 1 and old.Stato <> 0 then
		signal sqlstate '45001' set message_text = 'solo bevande nello stato ORDINATA
possono essere portate allo stato IN PREPARAZIONE ';
	elseif new.Stato = 2 and  old.Stato <> 1 then
		signal sqlstate '45001' set message_text = 'solo bevande nello stato IN
PREPARAZIONE possono essere portate allo stato PRONTA ';
   elseif new.Stato = 3 and  old.Stato <> 2 then
		signal sqlstate '45001' set message_text = 'solo bevande nello stato PRONTA
possono essere portate allo stato SERVITA ';
	end if;
END
```

verifica_stato_comanda_b

Il seguente trigger viene utilizzato quando viene segnata come ‘servita’ (3) una bevanda per verificare se è stato servito anche tutto il resto delle cose che erano state ordinate all’interno della stessa comanda, in caso positivo segna come ‘servita’ (1) tale comanda

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`verifica_stato_comanda_b`
AFTER UPDATE ON `Bevande_effettive` FOR EACH ROW
BEGIN
	if new.Stato = 3 then
		if not exists (select * from Pizze_effettive where Comanda = new.Comanda and
Stato != 3)
				and not exists (select * from Bevande_effettive where Comanda =
new.Comanda and Stato != 3) then
			update Comande set Servita = 1 where Numero = new.Comanda;
		end if;
	end if;
END
```

aggiorna_disponibilita_tavolo

Il seguente trigger va a segnare come non disponibile (0) il tavolo che viene assegnato al cliente

```
CREATE DEFINER = CURRENT_USER TRIGGER
`pizzeriaDB`.`aggiorna_disponibilita_tavolo` AFTER INSERT ON `Clienti` FOR EACH ROW
BEGIN
	UPDATE Tavoli
	SET Disponibilita = 0
	WHERE NumeroTavolo = new.Tavolo;
END
```

nuova_comanda

Il seguente trigger verifica prima dell'inserimento di una nuova comanda che essa faccia riferimento ad un tavolo che è occupato da commensali, cioè a cui è stato assegnato un cliente.
Nel caso non lo fosse non sarà possibile registrare la comanda.

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`nuova_comanda` BEFORE
INSERT ON `Comande` FOR EACH ROW
BEGIN
	if not exists (
			SELECT *
			FROM Clienti
			WHERE Tavolo = new.Tavolo) then

		signal sqlstate '45002' set message_text = 'tavolo non assegnato ad alcun cliente';
	end if;
END
```

decremento_ingredienti_c

Quando viene ordinata una pizza (e quindi effettuato un inserimento nella tabella *Pizze_effettive*) è necessario andare a decrementere di 1 unità la quantità presente in magazzino di tutti gli ingredienti utilizzati per condire tale pizza. In questo modo oltre ad utilizzare una gestione informatizzata del magazzino posso verificare se la pizza è disponibile, cioè se sono disponibili tutti i suoi ingredienti.

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`decremento_ingredienti_c`
BEFORE INSERT ON `Pizze_effettive` FOR EACH ROW
BEGIN
  UPDATE Ingredienti
	SET Quantita = Quantita - 1
	WHERE Nome IN (
				SELECT Ingrediente
				FROM Condimenti
				WHERE Pizza = new.NomePizza);
END
```

aggiornamento_stato_comanda_p

Il seguente trigger aggiorna lo stato della comanda quando viene ordinata una nuova pizza, questo viene effettuato come già introdotto per il trigger ‘*aggiornamento_stato_comanda_b’*

```
CREATE DEFINER = CURRENT_USER TRIGGER
`pizzeriaDB`.`aggiornamento_stato_comanda_p` AFTER INSERT ON `Pizze_effettive` FOR
EACH ROW
BEGIN
        UPDATE Comande
        SET Prezzo = Prezzo + (select Prezzo from Pizze where Nome = new.NomePizza)
        WHERE Numero = new.Comanda;

   if (SELECT Servita
        FROM Comande
        WHERE Numero = new.Comanda) then

        UPDATE Comande
        SET Servita = 0
        WHERE Numero = new.Comanda;

   end if;
END
```

verifica_stato_pizza

Il seguente trigger è simile a quello utilizzato per l'aggiornamento dello stato delle bevande (‘*verifica_stato_bevanda*’), ma questa volta si dovrà effettuare un controllo in più poiché quando viene richiesta l'ordinazione di una pizza inizialmente il suo stato viene settato per default a *null* (per indicare che la pizza è ‘in fase di ordinazione’) quindi solo da questo stato si potrà passare allo stato ‘ordinata’ (0).

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`verifica_stato_pizza`
BEFORE UPDATE ON `Pizze_effettive` FOR EACH ROW
BEGIN
   if new.Stato = 0 and old.Stato is not null then
            signal sqlstate '45003' set message_text = 'solo pizze in fase di ordinazione
                                                possono essere portate allo stato ORDINATA';
      elseif new.Stato = 1 and (old.Stato <> 0 or old.Stato is null) then
            signal sqlstate '45003' set message_text = 'solo pizze nello stato ORDINATA
                                                possono essere portate allo stato IN PRPEARAZIONE ';
      elseif new.Stato = 2 and  (old.Stato <> 1 or old.Stato is null) then
            signal sqlstate '45003' set message_text = 'solo pizze nello stato IN
                                    PREPARAZIONE possono essere portate allo stato PRONTA';
      elseif new.Stato = 3 and  (old.Stato <> 2 or old.Stato is null) then
            signal sqlstate '45003' set message_text = 'solo pizze nello stato PRONTA possono
                                                essere portate allo stato SERVITA ';
      end if;
END
```

verifica_stato_comanda_p

Il seguente trigger effettua lo stesso controllo del trigger '*verifica_stato_comanda_b*'

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`verifica_stato_comanda_p`
AFTER UPDATE ON `Pizze_effettive` FOR EACH ROW
BEGIN
	if new.Stato = 3 then
		if not exists (select * from Pizze_effettive where Comanda = new.Comanda and
Stato != 3)
				and not exists (select * from Bevande_effettive where Comanda =
new.Comanda and Stato != 3) then
			update Comande set Servita = 1 where Numero = new.Comanda;
		end if;
	end if;
END
```

comanda_servita

Il seguente trigger effettua un controllo prima dell'inserimento di un nuovo scontrino, ciò che verifica è che la comanda relativa al tavolo per cui viene richiesta la stampa dello scontrino sia stata completamente servita.

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`comanda_servita` BEFORE INSERT ON `Scontrini` FOR EACH ROW
BEGIN
	declare var_servita tinyint(1);
	declare msg varchar(100);

	SELECT Servita
	FROM Comande
	WHERE Tavolo = new.Tavolo INTO var_servita;

	if var_servita = 0 then
		set msg = concat('comanda ancora attiva, impossibile stampare lo scontrino del tavolo: ', new.Tavolo);
		signal sqlstate '45004' set message_text = msg;
	end if;
END
```

cancellazione_comanda

Il seguente trigger va ad eliminare la comanda relativa al tavolo per cui è stato stampato lo scontrino e di conseguenza anche tutte le pizze effettive (con le eventuali aggiunte) e le bevande effettive appartenenti ad essa, questo perché viene usata la clausola ON DELETE CASCADE per le foreign key relative al numero di comanda

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`cancellazione_comanda` AFTER INSERT ON `Scontrini` FOR EACH ROW
BEGIN
	delete from Comande where Tavolo = new.Tavolo;
END
```

scontrini_da_pagare

Il seguente trigger va a verificare che quando viene richiesto il pagamento di uno scontrino questo si abbia l'attributo 'Pagato' = 0

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`scontrini_da_pagare`
BEFORE UPDATE ON `Scontrini` FOR EACH ROW
BEGIN
	if (SELECT Pagato
		FROM Scontrini
		WHERE DataOra = new.DataOra) then

		signal sqlstate '45005' set message_text = "Lo stato di questo scontrino e':
PAGATO";
	end if;
END
```

pagamento_scontrino

Come detto prima al momento della stampa di uno scontrino viene eliminata la comanda relativa al tavolo in questione in modo tale da impedire di effettuare nuove ordinazioni che altrimenti verrebbero “perse” nel conteggio del prezzo da pagare. Un cliente potrebbe però decidere di ordinare ancora qualcosa anche dopo la stampa dello scontrino e prima del pagamento di esso ed in questo caso verrebbe registrata una nuova comanda per il tavolo che necessiterà poi della stampa e del pagamento di un nuovo scontrino.

Per questo motivo ci si potrebbe trovare nella situazione in cui si abbiano più scontrini da pagare per lo stesso tavolo oppure uno scontrino da pagare ed una comanda, e se non si facesse una verifica al momento del pagamento dello scontrino verebbe liberato il tavolo ed eliminato il cliente in modo incorretto.

Quindi dopo aver registrato il pagamento di uno scontrino e aver quindi aggiornato l’attributo ‘Pagato’ settandolo a 1, se non sono presenti altri scontrini da pagare o comande per quel tavolo, verrà eliminato dalla base di dati il cliente che era stato assegnato a tale tavolo poiché assumo che non è rilevante tenere memoria di tutti i clienti che sono stati nella pizzeria.
A questo punto viene settata a 1 la disponibilità del tavolo (non è più occupato e può essere assegnato ad un nuovo cliente).
Inoltre viene effettuato un altro controllo per quanto riguarda i turni del cameriere, questo poiché al termine del proprio turno al cameriere saranno tolti tutti i tavoli a cui è associato, ad eccezione di quelli in cui devono ancora essere pagati e/o stampati gli scontrini. Al momento del pagamento dello scontrino quindi anche tale tavolo potrà essere tolto al cameriere.

(*Vedere pagina successiva*)

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`pagamento_scontrino`
AFTER UPDATE ON `Scontrini` FOR EACH ROW
BEGIN
	if not exists (
			SELECT *
			FROM Scontrini
			WHERE Tavolo = new.Tavolo AND Pagato = 0)
		and not exists (
			SELECT *
			FROM Comande
			WHERE Tavolo = new.Tavolo) then

		delete from Clienti where Tavolo = new.Tavolo;

		UPDATE Tavoli
		SET Disponibilita = 1
		WHERE NumeroTavolo = new.Tavolo;

		if not exists (	SELECT *
				FROM Turni_Camerieri INNER JOIN Turni ON Turno = Nome
				WHERE Data = CURDATE()
					AND CURTIME() BETWEEN OraInizio AND OraFine
					AND (NomeC, CognomeC) = (
						SELECT NomeC, CognomeC
						FROM Tavoli
						WHERE NumeroTavolo = new.Tavolo))  then

			UPDATE Tavoli
			SET NomeC = NULL, CognomeC = NULL
			WHERE NumeroTavolo = new.Tavolo;
		end if;
	end if;
END
```

modifica_cameriere

Il seguente trigger effettua diverse verifiche riguardanti l'associazione di un cameriere ad un tavolo (inizialmente viene fatto il controllo sulla disponibilità per essere sicuri che l'update sia relativo al cameriere associato al tavolo e non alla sua disponibilità che viene ad esempio modificata quando viene assegnato/rimosso un cliente)

- come prima cosa verifico che non ci siano comande relative al tavolo in questione poiché in tal caso non può essere rimosso o cambiato il cameriere associato
- poi vado a verificare che non esistano scontrini ancora da pagare per tale tavolo poiché un cameriere può essere rimosso da un tavolo solo se la comanda è inattiva (cioè se è stato pagato lo scontrino) devo fare quest'ulteriore controllo poiché potrebbe essere che lo scontrino è stato stampato (e quindi la comanda cancellata), ma non ancora pagato.
- nel caso in cui si va ad associare un cameriere al tavolo (*nome e cognome diversi da null*):
  - il cameriere deve essere di turno nell'istante in cui viene richiesta l'associazione
  - il tavolo deve essere in uso nell'istante in cui viene richiesta l'associazione

(*Vedere pagina successiva*)

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`modifica_cameriere`
BEFORE UPDATE ON `Tavoli` FOR EACH ROW
BEGIN
	if old.Disponibilita = new.Disponibilita then
		if exists (
			SELECT *
			FROM Comande
			WHERE Tavolo = new.NumeroTavolo) then
		  signal sqlstate '45006' set message_text = 'sono presenti comande attive per
                                                                    questo tavolo';
		elseif exists (
				SELECT *
				FROM Scontrini
				WHERE Tavolo = new.NumeroTavolo AND Pagato = 0) then
		  signal sqlstate '45006' set message_text = 'deve ancora essere pagato lo scontrino
                                                            per questo tavolo';
		elseif new.NomeC is not null AND new.CognomeC is not null then
			if not exists (
				SELECT *
				FROM Turni_Camerieri INNER JOIN Turni
					ON Turni_Camerieri.Turno = Turni.Nome
				WHERE NomeC = new.NomeC
					AND CognomeC = new.CognomeC
					AND current_date = Turni_Camerieri.Data
					AND current_time between Turni.OraInizio
                                                        AND Turni.OraFine ) then
				signal sqlstate '45006' set message_text = 'cameriere non di turno';
			elseif not exists (
				SELECT *
				FROM Turni_Tavoli INNER JOIN Turni
					ON Turni_Tavoli.Turno = Turni.Nome
				WHERE Tavolo = new.NumeroTavolo
					AND current_date = Turni_Tavoli.Data
					AND current_time between Turni.OraInizio
                                                        AND Turni.OraFine) then
				signal sqlstate '45006' set message_text = 'tavolo non in uso';
			end if;
		end if;
	end if;
END
```

ore_di_lavoro

Il seguente trigger verifica la correttezza delle fasce orarie che il manager definisce per i turni, si è stabilito che in questo sistema ci siano turni fissi da al più 8 ore quindi OraFine non può essere maggiore di OraInizio + 8 ore

```
CREATE DEFINER = CURRENT_USER TRIGGER `pizzeriaDB`.`ore_di_lavoro` BEFORE
INSERT ON `Turni` FOR EACH ROW
BEGIN
	if new.OraFine > new.OraInizio + interval 8 hour then
		signal sqlstate '45007' set message_text = 'Gli orari inseriti non sono validi, i turni
devono essere al massimo di 8 ore';
	end if;
END
```

Eventi

```
set global event_scheduler = on;
```

rimuovi_vecchi_turni

Il seguente evento viene istanziato in fase di configurazione del sistema e, a partire dalle 00:00:00 della data in cui viene istanziato, va ogni giorno ad eliminare i turni dei camerieri e dei tavoli dei giorni precedenti

```
drop event if exists rimuovi_vecchi_turni;
delimiter |
create event if not exists rimuovi_vecchi_turni
on schedule every 1 day
starts timestamp(curdate(), '00:00:00')
on completion preserve
comment 'Cancella i turni dei giorni passati'
do begin
        delete from Turni_Tavoli where Data < curdate();
        delete from Turni_Camerieri where Data < curdate();
end |
delimiter ;
```

fine_turno_cameriere

Il seguente evento è stato introdotto per permettere la rimozione dei tavoli ad un cameriere che finisce il proprio turno, questo però può essere effettuato solo nel caso in cui il tavolo non abbia ancora una comanda attiva.
In supporto viene utilizzato il trigger '*pagamento_scontrino'* che va a togliere il tavolo al cameriere associato subito dopo il pagamento dell'ultimo scontrino relativo a tale tavolo nel caso in cui il cameriere non è più di turno.
Questo evento viene istanziato alle 00:00:00 della data in cui viene configurato il sistema ed eseguito ogni ora poiché qualunque siano le fasce orarie stabilite dal manager per i turni esse avranno comunque OraInizio ed OraFine della forma [HH:00:00]

(*Vedere pagina successiva*)

```
drop event if exists fine_turno_cameriere;
delimiter |
create event if not exists fine_turno_cameriere
on schedule every 1 hour
starts timestamp(curdate(), '00:00:00')
on completion preserve
comment 'Togli i tavoli ai camerieri che che hanno terminato il loro turno'
do begin

declare done int default false;
declare var_tavolo int unsigned;

declare cur cursor for
        select NumeroTavolo
        from Tavoli
        where NomeC is not null and CognomeC is not null
                and NumeroTavolo not in (select Tavolo from Comande)
                and NumeroTavolo not in (select Tavolo from Scontrini where Pagato = 0);

declare continue handler for not found set done = true;

open cur;
read_loop: loop
        fetch cur into var_tavolo;
        if done then
                leave read_loop;
        end if;

        UPDATE Tavoli
        SET  NomeC = NULL, CognomeC = NULL
        WHERE NumeroTavolo = var_tavolo and (NomeC, CognomeC) not in (
                                SELECT NomeC, CognomeC
                                FROM Turni_Camerieri JOIN Turni ON Turno = Nome
                                WHERE Data = CURDATE()
                                        AND (CURTIME() BETWEEN OraInizio AND OraFine));
end loop;
close cur;
END |
delimiter ;
```

Viste

tavoli_da_assegnare_a_clienti

La seguente vista viene utilizzata al momento della registrazione di un cliente poiché rende più semplice e immediata la visualizzazione dei tavoli che possono essere assegnati ai clienti. Mostra infatti, per ogni tavolo non occupato (disponibile) e attualmente in uso (di turno), il numero del tavolo e il numero di posti che tale tavolo mette a disposizione

```
CREATE VIEW `tavoli_da_assegnare_a_clienti` AS
   SELECT  NumeroTavolo AS NumTavolo,
           NumeroPosti AS NumPosti
   FROM  Tavoli
   WHERE  Disponibilita = 1
           AND pizzeriaDB.Tavoli.NumeroTavolo IN (
                       SELECT Tavolo
                       FROM Turni_Tavoli JOIN Turni
                               ON Turni_Tavoli.Turno = Turni.Nome
                       WHERE curtime() BETWEEN OraInizio AND OraFine)
```

cosa_servire

La seguente vista viene utilizzata per semplificare l'operazione di visualizzazione, da parte del cameriere, dei prodotti pronti per essere serviti

```
CREATE VIEW `cosa_servire` AS
        SELECT Comanda, Id, NomePizza AS NomeProdotto
        FROM Pizze_effettive
        WHERE Stato = 2

        UNION

        SELECT Comanda, Id, NomeBevanda AS NomeProdotto
        FROM Bevande_effettive
        WHERE Stato = 2;
```

Stored Procedures e transazioni

Inizio mostrando alcune funzioni che ho utilizzato come supporto all'esecuzione di alcune procedure

trova_nome_utente

La seguente funzione prende come parametro in input lo username ed estrae il nome dell'utente, questo viene fatto poiché si è assunto che tutti giu utilizzatori della base di dati dovranno autenticarsi utilizzando uno username della forma *"nome_cognome"*

```
CREATE FUNCTION `trova_nome_utente`(username varchar(91))
RETURNS varchar(45)
DETERMINISTIC
BEGIN
	declare nomeUtente varchar(45);
	select substring_index(username, '_', 1) into nomeUtente;
return nomeUtente;
END
```

trova_cognome_utente

Come la precedente ma estrae il cognome dell'utente

```
CREATE FUNCTION `trova_nome_utente`(username varchar(91))
RETURNS varchar(45)
DETERMINISTIC
BEGIN
	declare nomeUtente varchar(45);
	select substring_index(username, '_', 1) into nomeUtente;
return nomeUtente;
END
```

verifica_cameriere

La seguente funzione va a verificare se l'utente che ha come username il primo parametro in input alla funzione è il cameriere associato al tavolo passato come secondo parametro in input, tale funzine restituisce 1 in caso positivo, 0 altrimenti

```
CREATE FUNCTION `verifica_cameriere` (username varchar(91), tavolo int unsigned)
RETURNS TINYINT(1)
BEGIN
	declare var_nomeUtente varchar(45);
	declare var_cognomeUtente varchar(45);
	declare var_nomeC varchar(45);
	declare var_cognomeC varchar(45);
	set var_nomeUtente = trova_nome_utente(username);
	set var_cognomeUtente = trova_cognome_utente(username);

	SELECT NomeC, CognomeC
	FROM Tavoli
	WHERE NumeroTavolo = tavolo
	INTO var_nomeC, var_cognomeC;

	if (var_nomeUtente, var_cognomeUtente) <> (var_nomeC, var_cognomeC) or
(var_nomeC                                                    is null and
var_cognomeC is null) then
		return 0;
	else
		return 1;
	end if;
END
```

verifica_cameriere

La seguente funzione è simile alla precedente, ma questa volta vado a verificare che l'utente con lo username passato in input come primo parametro sia il cameriere che ha registrato la comanda, a differenza di quanto viene fatto nella funzione precedente qui non viene fatto il controllo su possibili valori NULL perché se esiste una comanda allora quetsa dovrà per forza essere stata presa dal cameriere associato al tavolo cui la comanda fa riferimento.

```
CREATE FUNCTION `verifica_cameriere_ordinazione` (username varchar(91), comanda int
unsigned)
RETURNS TINYINT(1)
BEGIN
	declare var_nomeUtente varchar(45);
	declare var_cognomeUtente varchar(45);

	set var_nomeUtente = trova_nome_utente(username);
	set var_cognomeUtente = trova_cognome_utente(username);

	if (var_nomeUtente, var_cognomeUtente) <> (
				SELECT NomeC, CognomeC
				FROM Tavoli JOIN Comande
					ON Tavoli.NumeroTavolo = Comande.Tavolo
				WHERE Comande.Numero = comanda) then
		return 0;
	else
		return 1;
	end if;
END
```

login

La seguente procedura viene utilizzata per effettuare il login e restutuisce il ruolo dell'utente che ha effettuato l'autenticazione

```
CREATE PROCEDURE `login` (in var_username varchar(91), in var_password varchar(45), out var_ruolo int)
BEGIN
	declare var_ruolo_utente ENUM('Manager', 'Cameriere', 'Pizzaiolo', 'Barista');

	SELECT Ruolo
	FROM Utenti
	WHERE Username = var_username
		AND Password = MD5(var_password) INTO var_ruolo_utente;

		if var_ruolo_utente = 'Manager' then
			set var_ruolo = 1;
		elseif var_ruolo_utente = 'Cameriere' then
			set var_ruolo = 2;
		elseif var_ruolo_utente = 'Pizzaiolo' then
			set var_ruolo = 3;
		elseif var_ruolo_utente = 'Barista' then
			set var_ruolo = 4;
		else
			set var_ruolo = 5;
		end if;
END
```

**** ho assunto che non si verifichi mai la situazione in cui più manager lavorino contemporaneamente e quindi eseguano azioni concorrenti ****

crea_turno

La seguente procedura viene utilizzata dal manager per definire le fasce orarie dei turni di lavoro.
I parametri in input sono: il nome che viene associato alla fascia oraria, l'orario di inizio e l'orario di fine.

```
CREATE PROCEDURE `crea_turno` (in var_nomeTurno VARCHAR(45), in var_oraInizio TIME, in var_oraFine TIME)
BEGIN
	insert into Turni values(var_nomeTurno, var_oraInizio, var_oraFine);
END
```

crea_turno_cameriere

La seguente procedura viene utilizzata dal manager per definire quali camerieri lavorano in quali turni.
I parametri in ingresso sono: il nome e il cognome del cameriere, il nome del turno che indica la fascia oraria e la data in cui il cameriere dovrà lavorare

```
CREATE PROCEDURE `crea_turno_cameriere` (in var_nomeCameriere varchar(45), in var_cognomeCameriere varchar(45), in var_turno varchar(45), in var_data date)
BEGIN
	insert into Turni_Camerieri
		values(var_nomeCameriere, var_cognomeCameriere, var_turno, var_data);
END
```

crea_turno_tavolo

La seguente procedura viene utilizzata dal manager per definire quali tavoli sono utilizzati in quali turni.

I parametri in ingresso sono: il numero del tavolo, il nome del turno che indica la fascia oraria e la data in cui il cameriere dovrà lavorare

```
CREATE PROCEDURE `crea_turno_tavolo` (in var_tavolo int unsigned, in var_turno
varchar(45), in var_data date)
BEGIN
	insert into Turni_Tavoli
		values (var_tavolo, var_turno, var_data);
END
```

associa_cameriere_tavolo

La seguente procedura viene utilizzata dal manager per associare un cameriere ad un tavolo.

I parametri in ingresso sono il nome e il cognome del cameriere ed il numero del tavolo.

È importante notare che l'inserimento di un nuovo tavolo e di un nuovo cameriere possono essere effettuati soltanto dal manager quindi le select su *Tavoli* e *Camerieri* non possono produrre letture sporche.

L'esecuzione di questa procedura va ad attivare il trigger *'modifica_cameriere'* e bisogna notare le seguenti cose:

- la select su *Comande* per verificare se esistano comande per quel tavolo potrebbe causare un inserimento fantasma se in modo concorente un cameriere va a registrare una nuova comanda per tale tavolo
- la select su *Scontrini* non può portare a letture sporche perché soltanto il manager può stampare uno scontrino o modificare l'attributo *Pagato*
- le altre select non vengono mai effettuate in questo caso poiché *NomeC* e *CognomeC* sono diversi da NULL

Viene quindi utilizzata la logica transazionale e il livello di isolamento è *serializable*

```
CREATE PROCEDURE `associa_cameriere_tavolo` (in var_tavolo int unsigned, in var_nomeC varchar(45), in var_cognomeC varchar(45))
BEGIN
	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;
	set transaction isolation level serializable;
	start transaction;

	if not exists (select * from Tavoli where NumeroTavolo = var_tavolo) then
		signal sqlstate '45011' set message_text = 'tavolo non presente nel sistema';

	elseif not exists (select * from Camerieri where Nome = var_nomeC and Cognome = var_cognomeC) then
		signal sqlstate '45011' set message_text = 'cameriere non presente nel sistema';

	end if;

	UPDATE Tavoli
	SET NomeC = var_nomeC, CognomeC = var_cognomeC
	WHERE NumeroTavolo = var_tavolo;

   commit;
END
```

togli_cameriere_tavolo

La seguente procedura viene utilizzata dal manager per eliminare l'associazione che era stata precedentemente effettuata tra un cameriere ed un tavolo.

I parametri in ingresso sono il nome e il cognome del cameriere ed il numero del tavolo.

Anche in questo caso viene utilizzato come livello di isolamento *serializable* poiché va sempre effettuato il controllo sull'eventuale presenza di comande che fanno riferimento al tavolo in questione, così come veniva fatto per l'associazione di un cameriere ad un tavolo.

```
CREATE PROCEDURE `togli_cameriere_tavolo` (in var_tavolo int unsigned, in var_nomeC
varchar(45), in var_cognomeC varchar(45))
BEGIN
	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;
	set transaction isolation level serializable;
	start transaction;

	if not exists (select * from Tavoli where NumeroTavolo = var_tavolo) then
		signal sqlstate '45022' set message_text = 'tavolo non presente nel sistema';
	elseif not exists (select * from Camerieri where Nome = var_nomeC and Cognome =
                                                                    var_cognomeC) then
		signal sqlstate '45022' set message_text = 'cameriere non presente nel sistema';

	elseif not exists (select * from Tavoli where NumeroTavolo = var_tavolo and NomeC =
                                                var_nomeC and CognomeC = var_cognomeC) then
		signal sqlstate '45022' set message_text = 'questo cameriere non era associato a
                                                                        questo tavolo';
	end if;
	UPDATE Tavoli
	SET NomeC = NULL, CognomeC = NULL
	WHERE NumeroTavolo = var_tavolo
		AND NomeC = var_nomeC
		AND CognomeC = var_cognomeC;
commit;
END
```

registra_cliente

La seguente procedura permette di registrare un nuovo cliente all'interno del sistema assegnandogli un tavolo, verrà quindi settato tale tavolo come occupato.
Viene utilizzata la vista *"tavoli_da_assegnare_a_clienti"* introdotta precedentemente.
I parametri in ingresso sono il nome e il cognome del cliente e il numero di commensali.
L'unico parametro in uscita è il numero del tavolo che viene assegnato al cliente.
È possibile utilizzare il livello di isolamento più basso, cioè *read uncommitted* poiché all'interno della pizzeria ho un solo manager per volta e soltanto lui potrà registrare i clienti e liberare un tavolo segnando come pagato lo scontrino relativo a tale tavolo, non sarà quindi possibile che si verifichi una condizione di lettura sporca effettuando la select su *tavoli_da_assegnare_a_clienti.*

```
CREATE PROCEDURE `registra_cliente` (in var_nomeCliente varchar(45), in
var_cognomeCliente varchar(45), in var_numeroCommensali int unsigned, out var_numeroTavolo
int unsigned)
BEGIN
	declare exit handler for sqlexception
	begin
		rollback;
		 resignal;
	end;
	set transaction isolation level read uncommitted;
	start transaction;

	SELECT NumTavolo
	FROM tavoli_da_assegnare_a_clienti
	WHERE var_numeroCommensali <= NumPosti
	LIMIT 1 INTO var_numeroTavolo;

	insert into Clienti
		values (var_nomeCliente, var_cognomeCliente, var_numeroTavolo,
                                                                    var_numeroCommensali);
	commit;
END
```

inserisci_pizza_menu

La seguente procedura viene utilizzata dal manager per inserire una nuova pizza nel menu.
Si decide di inserire la disponibilità di tale pizza con il valore di default, cioè a 0 poiché il menu viene definito una sola volta in un momento in cui i lavoratori della pizzeria non hanno ancora iniziato ad utilizzare il sistema, l'idea è la seguente:

al momento della definizione del menu verranno settati a 0 (default) sia il valore della quantità degli ingredienti sia quella delle bevande in magazzino e questi valori verranno poi incrementati in base al rifornimento che viene effettuato. Le pizze potranno quindi essere ordinate o meno in base alla presenza/assenza in magazzino di tutti gli ingredienti usati per condirla

```
CREATE PROCEDURE `inserisci_pizza_menu` (in var_nomePizza varchar(45), in var_prezzo decimal(4,2))
BEGIN
	insert into Pizze(Nome, Prezzo)
		values (var_nomePizza, var_prezzo);
END
```

inserisci_ingrediente_menu

```
CREATE PROCEDURE `inserisci_ingrediente_menu` (in var_nomeIngrediente varchar(45), in var_prezzo decimal(3,2))
BEGIN
	insert into Ingredienti(Nome, Prezzo)
		values (var_nomeIngrediente, var_prezzo);
END
```

inserisci_bevanda_menu

```
CREATE PROCEDURE `inserisci_bevanda_menu` (in var_nomeBevanda varchar(45), in
var_prezzo decimal(5,2))
BEGIN
        insert into Bevande(Nome, Prezzo)
                values (var_nomeBevanda, var_prezzo);
END
```

inserisci_condimento_pizza

La seguente procedura verrà utilizzata dal manager per definire quali sono gli ingredienti utilizzati per condire una pizza. Quest'operazione verrà effettuata inserendo un ingrediente per volta con la relativa pizza.

I parametri in ingresso sono: il nome della pizza e il nome dell'ingrediente

```
CREATE PROCEDURE `inserisci_condimento_pizza` (in var_pizza varchar(45), in
var_ingrediente varchar(45))
BEGIN
        insert into Condimenti
                values (var_pizza, var_ingrediente);
END
```

**** si suppone per semplicità che la quantità presente in magazzino degli ingredienti sia mantenuta in unità e non in peso, un'unità sarà infatti la quantità media di tale prodotto che viene utilizzata per condire una pizza e/o per effettuare un'aggiunta su di essa ****

aggiorna_quantita_ingrediente

La seguente procedura viene utilizzata dal manager quando arrivano in pizzeria i rifornimenti per un determinato ingrediente e permette di aggiornare la quantità presente in magazzino di tale ingrediente.

Potrebbe essere effettuata una *lettura sporca* ad esempio nel caso in cui venga letto un determinato valore per la quantità dell'ingrediente, ma tale valore non è stabile poiché era stato aggiornato da una transazione non ancora andata in commit e che quindi potrebbe pure andare in abort.

È importante inoltre notare che in modo concorrente il manager potrebbe voler aggiornare la quantità di un ingrediente e uno dei camerieri potrebbe voler inserire in una comanda una pizza che ha come condimento e/o usa come aggiunta tale ingrediente, tale inserimento porta un aggiornamento (decremento) al valore della quantità dell'ingrediente. Poiché è necessario mantenere le quantità in modo preciso per permettere una corretta gestione informatizzata del magazzino sarà necessario utilizzare il livello di isolamento *read committed*.

```
CREATE PROCEDURE `aggiorna_quantita_ingrediente` (in var_ingrediente varchar(45), in
var_rifornimento int unsigned)
BEGIN
	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;
	set transaction isolation level read committed;
	start transaction;

	if not exists (select * from Ingredienti where Nome = var_ingrediente) then
		signal sqlstate '45010' set message_text = 'ingrediente non presente nel sistema';
	end if;

	UPDATE Ingredienti
	SET Quantita = Quantita + var_rifornimento
	WHERE Nome = var_ingrediente;

   commit;
END
```

aggiorna_quantita_bevanda

La seguente procedura viene utilizzata dal manager quando arrivano in pizzeria i rifornimenti per una determinata bevanda e permette di aggiornare la quantità presente in magazzino di tale bevanda.
Utilizzo il livello di isolamento *read committed*, poiché ogni volta che un cameriere effettua l'ordinazione di una bevanda per una comanda verrà aggiornata la quantità disponibile in magazzino di tale bevanda, decrementandola di 1, quindi potrebbero verificarsi letture sporche dell'attuale quantità presente in magazzino della bevanda.
La select sulle bevande viene effettuata per verificare che la bevanda di cui si vuole aggiornare la quantità esista realmente all'interno del sistema, questo però non porta ad avere inserimenti fantasma poiché si assume che il menu venga definito una sola volta prima che tutti i lavoraori della pizzeria inizino ad utilizzare tale sistema.

```
CREATE PROCEDURE `aggiorna_quantita_bevanda` (in var_bevanda varchar(45), in
var_rifornimento int unsigned)
BEGIN
	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;
	set transaction isolation level read committed;
	start transaction;

	if not exists (select * from Bevande where Nome = var_bevanda) then
		signal sqlstate '45009' set message_text = 'bevanda non presente nel sistema';
	end if;

	UPDATE Bevande
	SET Quantita = Quantita + var_rifornimento
	WHERE Nome = var_bevanda;

   commit;
END
```

stampa_scontrino

La seguente procedura viene utilizzata dal manager quando viene richiesta la stampa di uno scontrino relativo ad un determinato tavolo.
L'unico parametro in ingresso è infatti il numero del tavolo.
Prima di tutto viene effettuata una verifica sullo stato della comanda relativa al tavolo per cui viene richiesto lo scontrino e quest'azione viene negata nel caso in cui la comanda non sia stata ancora completamente servita (questo viene fattto dal trigger '*comanda_servita*').
Al momento della stampa dello scontrino relativo ad un tavolo viene eliminata (tramite il trigger '*cancellazione_comanda*') la comanda che fa riferimento a tale tavolo e di conseguenza anche tutte le pizze effettive, le aggiunte e le bevande effettive appartenenti a tale comanda (in questo modo non sarà più possibile aggiungere cose alla comanda). Questo viene fatto in automatico all'eliminazione della comanda poiché viene utilizzata *cascade* come azione compensativa nel caso di cancellazione di una comanda.
Il livello di isolamento utilizzato è *repeatable read* poiché potrebbero verificarsi aggiornamenti fantasma nel caso in cui venga aggiornato l'attributo "Servita" di una comanda che fa riferimento al tavolo per il quale il manager, nello stesso istante, sta effettuando la stampa dello scontrino.
Ad esempio la comanda viene segnata come non servita (0) perché il cameriere ha inserito una nuova pizza/bevanda alla comanda oppure viene segnata come servita (1) perché il cameriere ha servito l'ultima pizza/bevanda che mancava in quella comanda.

(*Vedere pagina successiva*)

```
CREATE PROCEDURE `stampa_scontrino` (in var_tavolo int unsigned, out var_prezzo
decimal(6,2), out var_dataOra datetime)
BEGIN
   declare exit handler for sqlexception
        begin
                rollback;
                resignal;
        end;
        set transaction isolation level repeatable read;
        start transaction;

        SELECT Prezzo
        FROM Comande
        WHERE Tavolo = var_tavolo
        INTO var_prezzo;

        if var_prezzo = 0 then
                signal sqlstate '45021' set message_text = 'nessuna pizza/bevanda ordinata';
        end if;

        set var_dataOra = current_timestamp();

        insert into Scontrini(DataOra, Prezzo, Tavolo)
                value (var_dataOra, var_prezzo, var_tavolo);

   commit;
END
```

registra_pagamento_scontrino

La seguente procedura viene utilizzata dal manager quando viene richiesto il pagamento di uno scontrino (precedentemente stampato).
Il parametro in ingresso è l'attributo 'DataOra' dello scontrino, cioè la data e l'orario esatto in cui esso è stato stampato, viene utilizzato questo come parametro di input, anche se potrebbe essere complicato inserirlo, perché in questo modo siamo sicuri di discriminare i vari scontrini e si può immaginare questa situazione come nella vita reale:

> viene richiesto il conto, il manager stampa lo scontrino e in esso sarà specificato, oltre al prezzo da pagare, la data e l'orario in cui tale scontrino è stato stampato. Se il cliente non desidera pagare subito, si recherà dopo in cassa per pagare e a questo punto il manager prenderà lo scontrino e si occuperà lui di leggere la data e l'ora scritti nello scontrino e di inserirli per registrare il pagamento.

Prima della registrazione dell'avvenuto pagamento viene verificato, tramite il trigger '*scontrini_da_pagare*', se lo scontrino preso in considerazione deve ancora essere pagato.
È importante anche ricordare che al momento del pagamento dello scontrino relativo ad un tavolo, se non sono presenti altri scontrini da pagare o comande per il tavolo a cui fa riferimento lo scontrino, viene eliminato dal sistema il cliente a cui era stato assegnato tale tavolo, settata la disponibilità del tavolo a 1 ed infine verificato che il cameriere associato al tavolo che è stato appena liberato è ancora di turno, in caso contrario esso viene rimosso dal tavolo. Tutto questo viene fatto dal trigger '*pagamento_scontrino'*.
Il livello di isolamento utilizzato è *serializable* perché il cameriere potrebbe registrare una nuova comanda in modo concorrente al pagamento dello scontrino

(*Vedere pagina successiva*)

```
CREATE PROCEDURE `registra_pagamento_scontrino` (in var_scontrino datetime)
BEGIN
	declare var_pagato tinyint(1);

	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;
	set transaction isolation level serializable;
	start transaction;

	SELECT Pagato
   FROM Scontrini
   WHERE DataOra = var_scontrino
   INTO var_pagato;

   if (var_pagato = 1) or var_pagato is null then
		signal sqlstate '45016' set message_text = 'non esiste uno scontrino da pagare per
questo tavolo';
	end if;

	UPDATE Scontrini
   SET Pagato = 1
   WHERE DataOra = var_scontrino;

   commit;
END
```

inserisci_cameriere

La seguente procedura viene utilizzata dal manager per inserire un nuovo cameriere all'interno del sistema.

I parametri in input sono il nome e il cognome del cameriere.

```
CREATE PROCEDURE `inserisci_cameriere` (in var_nomeC varchar(45), in var_cognomeC varchar(45))
BEGIN
	insert into Camerieri
		values (var_nomeC, var_cognomeC);
END
```

inserisci_tavolo

La seguente procedura viene utilizzata dal manager per inserire un nuovo tavolo all'interno del sistema.

I parametri in input sono il numero del tavolo usato per identificarlo ed il numero di posti a sedere che esso mette a disposizione.

```
CREATE PROCEDURE `inserisci_tavolo` (in var_numeroTavolo int unsigned, in var_numeroPosti int unsigned)
BEGIN
	insert into Tavoli(NumeroTavolo, NumeroPosti)
		values (var_numeroTavolo, var_numeroPosti);
END
```

visualizza_entrata_giornaliera

La seguente procedura viene utilizzata dal manager per visualizzare le entrate di una determinata giornata.

L'unico parametro in ingresso è la data di cui si desidera visualizzare le entrate.

Non serve utilizzare come livello di isolamento *serializable* perché sia la visualizzazione delle entrate giornaliere sia la stampa dello scontrino sia il pagamento di esso possono essere effettuate solo dal managare e all'interno della pizzeria lavorerà soltanto un manager per volta, quindi non potranno verificarsi inserimenti fantasma calcolando il totale delle entrate per lo specifico giorno passato in input.

Quello che viene fatto è appunto andare a leggere e sommare tutti i prezzi degli scontrini che sono stati stampati nella data richiesta pagati, per poi restituire in output il risultato ottenuto.

```
CREATE PROCEDURE `visualizza_entrata_giornaliera` (in var_giorno date, out
var_entrata_giornaliera decimal(8,2))
BEGIN
	SELECT
		CASE
			WHEN SUM(Prezzo) IS NULL THEN 0
			ELSE SUM(Prezzo)
		END
	FROM Scontrini
	WHERE DATE(DataOra) = var_giorno AND Pagato = 1
	INTO var_entrata_giornaliera;
END
```

visualizza_entrata_mensile

La seguente procedura è simile alla precedente, però viene utilizzata per visualizzare le entrate di un determinato mese in un determinato anno.
I parametri in ingresso sono l'anno e il mese di cui si desidera visualizzare le entrate, mentre il parametro in uscita e la somma dei prezzi di tutti gli scontrini che sono stati stampati e pagati nel mese richiesto.

```
CREATE PROCEDURE `visualizza_entrata_mensile` (in var_anno year, in var_mese int, out
var_entrata_mensile decimal(9,2))
BEGIN
	SELECT
		CASE
			WHEN SUM(Prezzo) IS NULL THEN 0
			ELSE SUM(Prezzo)
		END
	FROM Scontrini
	WHERE YEAR(DataOra) = var_anno
		AND MONTH(DataOra) = var_mese
		AND Pagato = 1
	INTO var_entrata_mensile;
END
```

cancella_turno_cameriere

La seguente procedura è stata introdotta per permettere al manager di cancellare il turno di un cameriere. Questo perché si è assunto che il manager vada a definire una volta a settimana i turni per tutta la settimana successiva e quindi potrebbe verificarsi che un cameriere venga insertio in un determinato turno ma poi non possa realmente lavorare in quel giorno e in quelle facse orarie. Ricordo che tengo sempre conto del fatto che non lavorino più manager contemporaneamente e quindi non avverà mai in modo concorrente l'inserimento di più turni di lavoro per i camerieri.

```
CREATE PROCEDURE `cancella_turno_cameriere` (in var_nomeC varchar(45), in
var_cognomeC varchar(45), in var_turno varchar(45), in var_data date)
BEGIN
	if exists (SELECT *
				FROM Turni_Camerieri
				WHERE NomeC = var_nomeC
					AND CognomeC = var_cognomeC
					AND Turno = var_turno
					AND Data = var_data) then

		DELETE FROM Turni_Camerieri
		WHERE NomeC = var_nomeC
			AND CognomeC = var_cognomeC
			AND Turno = var_turno
			AND Data = var_data;
	else
		signal sqlstate '45012' set message_text = 'Non esiste questo turno tra i turni dei
                                                                    camerieri';
	end if;
END
```

visualizza_tavoli_occupati

La seguente procedura viene utilizzata dai camerieri per visualizzare quali dei tavoli a loro associati sono occupati.

Il parametro passato in input è lo username dell'utente che vuole eseguire questa stored procedure, questo verrà utilizzato per trovare il nome e il cognome dell'utente (mediante le funzioni '*trova_nome_utente*', '*trova_cognome_utente*') che serviranno per selezionare solo i tavoli a lui associati e visualizzare quali di questi sono occupati.

Il livello di isolamento usato è *read committed* per evitare possibili *letture sporche*, nel caso in cui ad esempio venga liberato (questo avviene con il pagamento dello scontrino) uno dei tavoli associati al cameriere che in quell'istante sta visualizzando quali tavoli sono occupati.

Inoltre la transazione è in sola lettura *(read only).*

```
CREATE PROCEDURE `visualizza_tavoli_occupati` (in var_username varchar(91))
BEGIN
        declare exit handler for sqlexception
        begin
                rollback;
                resignal;
        end;
        set transaction isolation level read committed;
   set transaction read only;
        start transaction;

        SELECT NumeroTavolo
        FROM Tavoli
        WHERE Disponibilita = 0
                AND NomeC = TROVA_NOME_UTENTE(var_username)
                AND CognomeC = TROVA_COGNOME_UTENTE(var_username);

   commit;
END
```

visualizza_tavoli_comande_servite

La seguente procedura viene utilizzata dai camerieri per visualizzare in quali dei tavoli a loro associati le relative comande sono state completamente servite.
L'unico parametro passato in input è lo username dell'utente che vuole eseguire questa stored procedure, questo verrà utilizzato per trovare il nome e il cognome dell'utente (mediante le funzioni '*trova_nome_utente*', '*trova_cognome_utente*') e selezionare così soltanto i tavoli a lui associati (se presenti).
Il livello di isolamento utilizzato è ***serializable*** poiché un'altra transazione potrebbe cancellare tuple che dovrebbero invece essere considerate da questa transazione, ad esempio contestualmente il manager potrebbe registrare il pagamento dello scontrino relativo ad un tavolo e un cameriere potrebbe visualizzare in quali dei tavoli a lui associati le comande sono state completamente servite (e il tavolo per cui viene richiesto il pagamento fa parte di questi). Infatti solo con questo livello di isolamento è possibile gestire i *range lock*.

```
CREATE PROCEDURE `visualizza_tavoli_comande_servite` (in var_username varchar(91))
BEGIN
        declare exit handler for sqlexception
        begin
                rollback;
                resignal;
        end;
        set transaction isolation level serializable;
        set transaction read only;
        start transaction;

        SELECT NumeroTavolo
        FROM Tavoli JOIN Comande ON Tavoli.NumeroTavolo = Comande.Tavolo
        WHERE Comande.Servita = 1
                AND NomeC = TROVA_NOME_UTENTE(var_username)
                AND CognomeC = TROVA_COGNOME_UTENTE(var_username);
   commit;
END
```

registra_comanda

La seguente procedura viene utilizzata dai camerieri per registrare una nuova comanda relativa ad un tavolo.
L'attributo "Servita" sarà settato a 0 (default) che indica che la comanda non è stata completamente servita, ugualmente anche "Prezzo" sarà a 0 (default) poiché questo valore verrà incrementato man mano che vengono inserite pizze (con eventuali aggiunte) e bevande sommando il loro prezzo.
Il parametro in ingresso è il numero del tavolo per cui viene richiesta la registrazione della comanda.
Il parametro in uscita è il numero di comanda, esso viene generato con l'autoincrement.

Per l'esecuzione di questa procedura si ricorre all'utilizzo della funzione '*verifica_cameriere'*, che verifica se colui che sta richiedendo la registrazione della comanda è il cameriere associato al tavolo.

Il livello di isolamento è *serializable* poiché prima di effettuare l'inserimento viene attivato il trigger *'nuova_comanda'* che va ad effettuare una select su *Comande* per verificare che ci sia un cliente associato al tavolo per cui si vuole registrare la comanda, dato che è il manager colui che si occupa della registrazione dei clienti e dell'assegnazione dei tavoli ad essi, potrebbe verficarsi che venga richiesto l'inserimento di un nuovo cliente ed in modo concorrente si richieda la registrazione della comanda per il medesimo tavolo da assegnare al cliente.
Se non venissse usato *serializable* come livello di isolamento e venisse prima schedulata la richiesta di registrazione della comanda e dopo la registrazione di un cliente con assegnazione al tavolo per cui è gia stata richiesta la comanda si verificherebbe un inserimento fantasma e la registrazione della comanda andrebbe in abort.

L'esecuzione della funzione '*verifica_cameriere'* potrebbe invece portare ad una lettura sporca poiché il manager potrebbe asscoiare/togliere un cameriere al tavolo in modo concorrente.
Infine viene usata l*ast_insert_id()* che è una funzione che restituisce l'ultimo id inserito per sessione, quindi questo non crea problemi per la concorrenza.

(*Vedere pagina successiva*)

```
CREATE PROCEDURE `registra_comanda` (in var_tavolo int unsigned, out var_numeroComanda
int unsigned, in var_username varchar(91))
BEGIN
	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;

	set transaction isolation level serializable;
	start transaction;

	if (verifica_cameriere(var_username, var_tavolo) = 0 ) then
		signal sqlstate '45015' set message_text = 'non sei il cameriere associato al tavolo
                                                    quindi non puoi registrare la comanda';
	end if;

	insert into Comande(Tavolo) value (var_tavolo);

	commit;
	set var_numeroComanda = last_insert_id();
END
```

ordina_bevanda

La seguente procedura viene utilizzata dai camerieri per ordinare una nuova bevanda effettiva per una determinata comanda.
I parametri in input sono: il nome della bevanda, e il numero della comanda.
L'Id di ogni nuova bevanda ordinata viene calcolato in base a se ci sono e quante sono le bevande già ordinate per quella comanda.
È importante ricordare che ogni volta che viene ordinata una nuova bevanda bisognerà effettuare le seguenti operazioni aggiuntive:

- decrementare di 1 unità la quantità presente in magazzino della bevanda che viene ordinata, questo viene fatto dal trigger *'decremento_bevanda'* introdotto prima e porta alla necessità di utilizzare come livello di isolamento *read committed* in modo tale da evitare eventuali letture sporche della quantità che dovrà essere decrementata nel caso in cui più transazioni cerchino di decrementare tale quantità in modo concorrente
- verificare che chi sta prendendo l'ordinazione è il cameriere associato al tavolo cui la comanda fa riferimento, viene utilizzata la funzione '*verifica_cameriere_ordinazione'* questa potrebbe portare ad una lettura sporca
- aggiornare lo stato della comanda a cui è stata aggiunta la bevanda settando a 0 (se prima era 1) l'attributo 'Servita' e sommando il prezzo della bevanda al prezzo attuale della comanda, questo viene fatto dal trigger *'aggiorna_stato_comanda_b'* , anche in questo caso potrebbero verficarsi letture sporche sul prezzo attuale della comanda

(*Vedere pagina successiva*)

```
CREATE PROCEDURE `ordina_bevanda` (in var_bevanda varchar(45), in var_comanda int
unsigned, in var_username varchar(91))
BEGIN
	declare var_idBevanda int unsigned;

	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;

	set transaction isolation level read committed;
	start transaction;

	if(verifica_cameriere_ordinazione(var_username, var_comanda) = 0) then
		signal sqlstate '45013' set message_text = 'non sei il cameriere associato al tavolo
		quindi non puoi prendere ordinazioni per questa comanda';
	end if;

	SELECT
		CASE
			WHEN MAX(Id) IS NULL THEN 1
			ELSE MAX(id) + 1
		END
	FROM Bevande_effettive
	WHERE Comanda = var_comanda INTO var_idBevanda;

	insert into Bevande_effettive(Id, NomeBevanda, Comanda)
		values (var_idBevanda, var_bevanda, var_comanda);

  commit;
END
```

ordina_pizza

La seguente procedura viene utilizzata dai camerieri per ordinare una nuova pizza effettiva per una determinata comanda.
I parametri in input sono: il nome della pizza, e il numero della comanda.
L'Id di ogni nuova pizza ordinata viene calcolato, come nel caso dell'ordinazione di una bevanda, in base a se ci sono e quante sono le pizze già ordinate per quella comanda e questo valore viene poi restituito in output perché sarà utilizzato (insieme al numero di comanda) per effettuare eventuali aggiunte alla pizza appena ordinata.
Anche in questo caso ci sono alcune operazioni aggiuntive da eseguire:

- decrementare di 1 la quantità presente in magazzino degli ingredienti che vengono usati per condire tale pizza, questo viene fatto dal trigger *'decremento_ingredienti_c'* e mi permette anche di verificare se la pizza è disponibile perché essendo "Quantita" un intero unsigned esso non potrà mai avere un valore inferiore allo zero, quindi se si prova a decrementare la quantità di un ingrediente che è gia a 0 mi viene restituito errore
- verificare che chi sta prendendo l'ordinazione è il cameriere associato al tavolo cui la comanda fa riferimento, viene utilizzata la funzione *'verifica_cameriere_ordinazione'*
- settare a 0 l'attributo 'Servita' della comanda a cui è stata aggiunta la pizza se prima era a 1, perché ci sarà questa nuova pizza che dovrà essere servita e sommare il prezzo della pizza al prezzo della comanda, questo viene fatto dal trigger *'aggiornamento_stato_comanda_p'*

Il livello di isolamento utilizzato è *read committed* in modo tale da evitare *letture sporche* durante il controllo della disponibilità della pizza (cioè la lettura delle quantità disponibili degli ingredienti usati per condirle) e l'aggiornamento della quantità degli ingredienti usati per condire la pizza ordinata.

(*Vedere pagina successiva*)

```
CREATE PROCEDURE `ordina_pizza` (in var_pizza varchar(45), in var_comanda int unsigned, in var_username varchar(91), out var_idPizza int unsigned)
BEGIN
	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;
	set transaction isolation level read committed;
	start transaction;

  if(verifica_cameriere_ordinazione(var_username, var_comanda) = 0) then
		signal sqlstate '45014' set message_text = 'non sei il cameriere associato al tavolo quindi non puoi prendere ordinazioni per questa comanda';
	end if;

	SELECT
		CASE
			WHEN MAX(Id) IS NULL THEN 1
			ELSE MAX(id) + 1
		END
	FROM Pizze_effettive
	WHERE Comanda = var_comanda INTO var_idPizza;

  insert into Pizze_effettive(Id, NomePizza, Comanda)
		values (var_idPizza, var_pizza, var_comanda);

  commit;
END
```

inserisci_aggiunta_pizza

La seguente procedura viene utilizzata dai camerieri per inserire una nuova aggiunta ad una determinata pizza effettiva già inserita precedentemente in una comanda.
I parametri in ingresso sono: il numero della comanda in cui è stata inserita la pizza effettiva, l'id della pizza effettiva per quella determinata comanda, il nome dell'ingrediente che si vuole usare come aggiunta.
Le operazioni aggiuntive da eseguire sono:

- verificare che la pizza a cui si vuole applicare l'aggiunta sia ancora in fase di ordinazione, questo viene fatto dal trigger *'verifica_fase_ordinazione'* e non crea problemi per la concorrenza perché solo il cameriere che sta prendendo quell'ordinazione potrà eaggiornare lo stato della pizza da *null* (in fase di ordinazione) a 0 (ordinata)
- decrementare di 1 la quantità presente in magazzino dell'ingrediente che si vuole utilizzare come aggiunta, questo viene fatto dal trigger '*decremento_ingrediente_a*' , come detto prima essendo "Quantita" un intero unsigned nel caso in cui vale 0 (cioè l'ingrediente non è disponibile) verrà direttamente restituito errore
- sommare il prezzo dell'aggiunta al prezzo della comanda, questo viene fatto dal trigger '*aggiorna_stato_comanda_a*' , questa volta però non si deve apportare alcuna modifica all'attributo "Servita" della comanda

Il livello di isolamento utilizzato è *read committed* in modo tale da evitare *letture sporche* durante la lettura della quantità disponibile e l'aggiornamento della quantità dell'ingrediente usato come aggiunta.

(*Vedere pagina successiva*)

```
CREATE PROCEDURE `inserisci_aggiunta_pizza` (in var_idPizza int unsigned, in var_comanda
int unsigned, in var_ingrediente varchar(45))
BEGIN
	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;
	set transaction isolation level read committed;
	start transaction;

	insert into Aggiunte
		values (var_idPizza, var_comanda, var_ingrediente);

   commit;
END
```

segna_pizza_ordinata

Questa procedura è stata introdotta perché quando un cameriere inserirà una pizza effettiva in una determinata comanda lo stato di tale pizza sarà settato di default a null, come per indicare che è in fase di ordinazione, in questo modo permetterò al cliente di richiedere delle aggiunte e il cameriere segnerà una pizza come "ordinata" (0) se il cliente non richiede di effettuare aggiunte su di essa o quando tutte le aggiunte richieste saranno già state applicate.
Questo permette anche al pizzaiolo di visualizzare le pizze da preparare solo quando la fase di ordinazione di tali pizze è stata completata.
I parametri in ingresso sono: l'Id della pizza e il numero della comanda in cui essa è stata inserita.
Una pizza che si trova nello stato null potrà essere portata solo allo stato 0 (questo controllo viene fatto dal trigger '*verifica_stato_pizza*') e quest'aggiornamento dello stato potrà essere effttuato solo ed esclusivamente dal cameriere che ha registrato la comanda alla quale appartiene la pizza presa in considerazione

```
CREATE PROCEDURE `segna_pizza_ordinata` (in var_idPizza int unsigned, in var_comanda int unsigned)
BEGIN
	UPDATE Pizze_effettive
	SET Stato = 0
	WHERE Id = var_idPizza
		AND Comanda = var_comanda;
END
```

visualizza_cosa_e_dove_servire

La seguente procedura viene utilizzata dai camerieri per visualizzare ciò che è pronto in relazione alle comande prese da loro e a quale tavolo servire ciò che è pronto.
Viene utilizzata la vista '*cosa_servire*' introdotta precedentemente.
Il livello di isolamento utilizzato è *repeatable read* in modo tale che se ad esempio un barista/pizzaiolo segna una bevanda/pizza come "Pronta" e in modo concorrente il cameriere esegue questa procedura potrà vedere tale bevanda/pizza tra le cose da servire una volta che la transazione che la segna come pronta sarà andata in commit (no letture sporche) e se dopo aver effettuato la prima select dalla vista '*cosa_servire'* vengono segnate come pronte altre pizze/bevande queste non verranno considerate dalla transazione.

(*Vedere pagina successiva*)

```
CREATE PROCEDURE `visualizza_cosa_e_dove_servire`(in var_username varchar(91))
BEGIN
	declare done int default false;
	declare var_comanda int unsigned;

	declare cur cursor for
			SELECT DISTINCT Comanda
			FROM cosa_servire;

	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;

	declare continue handler for not found set done = true;

	set transaction isolation level repeatable read;
	set transaction read only;
	start transaction;

	SELECT Numero, Tavolo
	FROM Comande
	WHERE Numero IN (SELECT Comanda
				FROM cosa_servire);
	open cur;
	set done = false;
	read_loop: loop
		fetch cur into var_comanda;
		if done then
			leave read_loop;
		end if;

		SELECT Id, NomeProdotto
		FROM cosa_servire
		WHERE Comanda = var_comanda;
	end loop;
   close cur;
commit;
END
```

segna_pizza_servita

La seguente procedura viene utilizzata dai camerieri quando servono una pizza.

I parametri in ingresso sono: l'Id della pizza e il numero della comanda.

È necessario effettuare un'operazione aggiuntiva per tenere aggiornato l'attributo "Servita" della comanda infatti oltre ad aggiornare lo stato della pizza effettiva si attiverà subito dopo l'aggiornamento il trigger '*verifica_stato_comanda_p*' che setterà ad 1 "Servita" se tutte le altre pizze/bevande della comanda sono state servite.

Il livello di isolamento utilizzato è *repeatable read* in modo tale da evitare letture sporche degli stati e aggiornamenti fantasma dovuti ai possibili aggiornamenti sullo stato effettuati dal pizzaiolo.

```
CREATE PROCEDURE `segna_pizza_servita` (in var_idPizza int unsigned, in var_comanda int
unsigned, in var_username varchar(91))
BEGIN
	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;
	set transaction isolation level repeatable read;
	start transaction;

	if(verifica_cameriere_ordinazione(var_username, var_comanda) = 0) then
		signal sqlstate '45020' set message_text = 'non sei il cameriere associato al tavolo
				quindi non puoi apportare quetsa modifica allo stato della pizza';
	end if;

	if not exists (select * from Pizze_effettive where Id = var_idPizza and Comanda =
							var_comanda) then
		signal sqlstate '45020' set message_text = 'non esiste questa pizza in questa
							comanda';
	end if;
	UPDATE Pizze_effettive
	SET Stato = 3
	WHERE Id = var_idPizza
		AND Comanda = var_comanda;
commit;
END
```

segna_bevanda_servita

Per la seguente procedura valgono gli stessi concetti espressi per *"segna_pizza_servita"*, in questo caso il trigger che si occuperà di mantenere coerente lo stato della comanda è '*verifica_stato_comanda_b*'

```
CREATE PROCEDURE `segna_bevanda_servita` (in var_idBevanda int unsigned, in
var_comanda int unsigned, in var_username varchar(91))
BEGIN
	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;
	set transaction isolation level repeatable read;
	start transaction;

	if(verifica_cameriere_ordinazione(var_username, var_comanda) = 0) then
		signal sqlstate '45018' set message_text = 'non sei il cameriere associato al tavolo
				quindi non puoi apportare quetsa modifica allo stato della pizza';
	end if;

	if not exists (select * from Bevande_effettive where Id = var_idBevanda and Comanda =
                                                                    var_comanda) then
		signal sqlstate '45018' set message_text = 'non esiste questa bevanda in questa
                                                                        comanda';
	end if;

	UPDATE Bevande_effettive
	SET Stato = 3
	WHERE Id = var_idBevanda
		AND Comanda = var_comanda;

   commit;
END
```

visualizza_pizze_da_preparare

La seguente procedura viene utilizzata dai pizzaioli per visualizzare le pizze da preparare in ordine di ricezione della comanda.

È necessario utilizzare *repeatable read* come livello di isolamento poiché altrimenti più pizzaioli potrebbero visualizzare e prendere in carico le stesse pizze, in questo modo verrebbero quindi preparate più volte, oppure potrebbe essere settata come in preparazione una pizza che è stata ordinata concorrentemente alla visualizzazione delle pizze da preparare e quindi non letta dalla prima select.

Quello che fa è effettuare prima la select delle pizze effettive e poi l'update dello stato di tali pizze portandolo da 0 a 1, al momento dell'update la transazione che non viene schedulata per prima rimarrà in attesa del commit della prima, ma una volta che essa effettuerà il commit lo stato delle pizze che ha vsiualizzato non sarà più 0 ma 1 e quindi la transazione che prima era in attesa verrà sbloccata ma non riuscirà a fare l'update (questo controllo viene effettuato dal trigger '*verifica_stato_pizza*') riscontrando un eccezione ed andando quindi in rollback.

Utilizzando questo livello di isolamento invece di serializable si permette ai camerieri di ordinare delle pizze anche in modo concorrente alla visualizzazione, da parte del pizzaiolo, delle pizze da preparare.

```
CREATE PROCEDURE `visualizza_pizze_da_preparare`()
BEGIN
	declare done int default false;
	declare var_comanda int unsigned;
	declare var_idPizza int unsigned;

	declare cur cursor for
			SELECT Comanda,    Id
			FROM Pizze_effettive
			WHERE Stato = 0
			ORDER BY Comanda, Id;
	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;
	declare continue handler for not found set done = true;
```

continua alla pagina successiva

```
        set transaction isolation level repeatable read;
        start transaction;

        SELECT Comanda,    Id, NomePizza
        FROM Pizze_effettive
        WHERE Stato = 0
        ORDER BY Comanda, Id;

        open cur;
        set done = false;
        read_loop: loop
                fetch cur into var_comanda, var_idPizza;
                if done then
                        leave read_loop;
                end if;

                SELECT Ingrediente
                FROM Aggiunte
                WHERE Comanda = var_comanda AND IdPizza = var_idPizza;
        end loop;
        close cur;

        open cur;
        set done = false;
        read_loop: loop
                fetch cur into var_comanda, var_idPizza;
                if done then
                        leave read_loop;
                end if;
                UPDATE Pizze_effettive
                SET Stato = 1
                WHERE Id = var_idPizza AND Comanda = var_comanda;
        end loop;
        close cur;
commit;
END
```

viusalizza_bevande_da_preparare

La seguente procedura viene utilizzata dai baristi per visualizzare le bevande da preparare in ordine di ricezione della comanda.
Il funzionamento è uguale a quello utilizzato per la visualizzazione delle pizze da preparare soltanto che in questo caso, avendo assunto che la pizzeria utilizzi solo bevande insdustriali, il barista non dovrà visualizzare da quali ingredienti è composta, ma soltano il nome, l'id e la comanda.
Quindi come nel caso precedente il livello di isolamento utilizzato è *repeatable read*.

```
CREATE PROCEDURE `visualizza_bevande_da_preparare`()
BEGIN
	declare done int default false;

	declare var_comanda int unsigned;
	declare var_idBevanda int unsigned;

	declare cur cursor for
		SELECT Comanda, Id
		FROM Bevande_effettive
		WHERE Stato = 0
		ORDER BY Comanda;

	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;

	declare continue handler for not found set done = true;

	set transaction isolation level repeatable read;
	start transaction;

	SELECT Comanda, Id, NomeBevanda
	FROM Bevande_effettive
	WHERE Stato = 0
	ORDER BY Comanda;
```

continua alla pagina successiva

```
        open cur;
        read_loop: loop
                fetch cur into var_comanda, var_idBevanda;
                if done then
                        leave read_loop;
                end if;

                UPDATE Bevande_effettive
                SET Stato = 1
                WHERE Id = var_idBevanda AND Comanda = var_comanda;
        end loop;
        close cur;
commit;
END
```

segna_pizza_pronta

La seguente procedura viene utilizzata dai pizzaioli per segnare che una pizza è pronta e può quindi essere servita dai camerieri.

Il livello di isolamento utilizzato è ***read committed*** in modo tale da evitare letture sporche dello stato della pizza, infatti prima dell'update si attiverà il trigger '*verifica_stato_pizza*' che andrà a leggere lo stato della pizza e verificherà la fattibilità dell'aggiornamento.

```
CREATE PROCEDURE `segna_pizza_pronta` (in var_idPizza int unsigned, in var_comanda int unsigned)
BEGIN
	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;
	set transaction isolation level read committed;
	start transaction;

	if not exists (select * from Pizze_effettive where Id = var_idPizza and Comanda = var_comanda) then
		signal sqlstate '45019' set message_text = 'non esiste questa pizza in questa comanda';
	end if;

	UPDATE Pizze_effettive
	SET Stato = 2
	WHERE Id = var_idPizza
		AND Comanda = var_comanda;

	commit;
END
```

segna_bevanda_pronta

La seguente procedura viene utilizzata dai baristi per segnare che una bevanda è pronta e può quindi essere servita dai camerieri.

Il funzionamento di tale procedura è analogo a quello della procedura utilizzata per le pizze, il trigger che si attiverà in questo caso è '*verifica_stato_bevanda*'

```
CREATE PROCEDURE `segna_bevanda_pronta` (in var_idBevanda int unsigned, in
var_comanda int unsigned)
BEGIN
	declare exit handler for sqlexception
	begin
		rollback;
		resignal;
	end;
	set transaction isolation level read committed;
	start transaction;

	if not exists (select * from Bevande_effettive where Id = var_idBevanda and Comanda =
                                                                    var_comanda) then
		signal sqlstate '45017' set message_text = 'non esiste questa bevanda in questa
                                                                        comanda';
	end if;

	UPDATE Bevande_effettive
	SET Stato = 2
	WHERE Id = var_idBevanda
		AND Comanda = var_comanda;

	commit;
END
```

Appendice: Implementazione

Codice SQL per instanziare il database

```
SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS,
FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE,
SQL_MODE='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO
_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION';
-- -----------------------------------------------------
-- Schema pizzeriaDB
-- -----------------------------------------------------
DROP SCHEMA IF EXISTS `pizzeriaDB` ;
CREATE SCHEMA IF NOT EXISTS `pizzeriaDB` ;
USE `pizzeriaDB` ;

-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Ingredienti`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Ingredienti` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Ingredienti` (
  `Nome` VARCHAR(45) NOT NULL,
  `Quantita` INT UNSIGNED NOT NULL DEFAULT 0,
  `Prezzo` DECIMAL(3,2) NOT NULL DEFAULT 1,
  PRIMARY KEY (`Nome`))
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Pizze`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Pizze` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Pizze` (
  `Nome` VARCHAR(45) NOT NULL,
  `Prezzo` DECIMAL(4,2) NOT NULL,
  PRIMARY KEY (`Nome`))
ENGINE = InnoDB;
```

```
-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Bevande`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Bevande` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Bevande` (
  `Nome` VARCHAR(45) NOT NULL,
  `Quantita` INT UNSIGNED NOT NULL DEFAULT 0,
  `Prezzo` DECIMAL(5,2) NOT NULL,
  PRIMARY KEY (`Nome`))
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Turni`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Turni` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Turni` (
  `Nome` VARCHAR(45) NOT NULL,
  `OraInizio` TIME NOT NULL,
  `OraFine` TIME NOT NULL,
  PRIMARY KEY (`Nome`),
  UNIQUE INDEX `FasciaOraria_UNIQUE` (`OraInizio` ASC, `OraFine` ASC) VISIBLE)
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Camerieri`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Camerieri` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Camerieri` (
  `Nome` VARCHAR(45) NOT NULL,
  `Cognome` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`Nome`, `Cognome`))
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Tavoli`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Tavoli` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Tavoli` (
```

```
  `NumeroTavolo` INT UNSIGNED NOT NULL,
  `NumeroPosti` INT UNSIGNED NOT NULL,
  `Disponibilita` TINYINT(1) UNSIGNED NOT NULL DEFAULT 1,
  `NomeC` VARCHAR(45) NULL DEFAULT NULL,
  `CognomeC` VARCHAR(45) NULL DEFAULT NULL,
  PRIMARY KEY (`NumeroTavolo`),
  INDEX `fk_Tavoli_Cameriere_idx` (`NomeC` ASC, `CognomeC` ASC) VISIBLE,
  CONSTRAINT `fk_Tavoli_Cameriere`
    FOREIGN KEY (`NomeC` , `CognomeC`)
    REFERENCES `pizzeriaDB`.`Camerieri` (`Nome` , `Cognome`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Comande`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Comande` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Comande` (
  `Numero` INT UNSIGNED NOT NULL AUTO_INCREMENT,
  `Tavolo` INT UNSIGNED NOT NULL,
  `Prezzo` DECIMAL(6,2) NOT NULL DEFAULT 0,
  `Servita` TINYINT(1) UNSIGNED NOT NULL DEFAULT 0,
  PRIMARY KEY (`Numero`),
  UNIQUE INDEX `Tavolo_UNIQUE` (`Tavolo` ASC) VISIBLE,
  CONSTRAINT `fk_Comande_Tavolo`
    FOREIGN KEY (`Tavolo`)
    REFERENCES `pizzeriaDB`.`Tavoli` (`NumeroTavolo`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Scontrini`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Scontrini` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Scontrini` (
  `DataOra` DATETIME NOT NULL,
```

```
  `Prezzo` DECIMAL(6,2) NOT NULL,
  `Pagato` TINYINT(1) UNSIGNED NOT NULL DEFAULT 0,
  `Tavolo` INT UNSIGNED NOT NULL,
  PRIMARY KEY (`DataOra`),
  INDEX `fk_Scontrini_Tavolo_idx` (`Tavolo` ASC) VISIBLE,
  CONSTRAINT `fk_Scontrini_Tavolo`
    FOREIGN KEY (`Tavolo`)
    REFERENCES `pizzeriaDB`.`Tavoli` (`NumeroTavolo`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Pizze_effettive`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Pizze_effettive` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Pizze_effettive` (
  `Id` INT UNSIGNED NOT NULL,
  `Comanda` INT UNSIGNED NOT NULL,
  `NomePizza` VARCHAR(45) NOT NULL,
  `Stato` TINYINT(1) UNSIGNED NULL DEFAULT NULL,
  PRIMARY KEY (`Id`, `Comanda`),
  INDEX `fk_PizzeEffettive_Comanda_idx` (`Comanda` ASC) VISIBLE,
  INDEX `fk_PizzeEffettive_Pizza_idx` (`NomePizza` ASC) VISIBLE,
  CONSTRAINT `fk_PizzeEffettive_Comanda`
    FOREIGN KEY (`Comanda`)
    REFERENCES `pizzeriaDB`.`Comande` (`Numero`)
    ON DELETE CASCADE
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_PizzeEffettive_Pizza`
    FOREIGN KEY (`NomePizza`)
    REFERENCES `pizzeriaDB`.`Pizze` (`Nome`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

```
-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Bevande_effettive`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Bevande_effettive` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Bevande_effettive` (
  `Id` INT UNSIGNED NOT NULL,
  `Comanda` INT UNSIGNED NOT NULL,
  `NomeBevanda` VARCHAR(45) NOT NULL,
  `Stato` TINYINT(1) UNSIGNED NOT NULL DEFAULT '0',
  PRIMARY KEY (`Id`, `Comanda`),
  INDEX `fk_BevandeEffettiva_Comanda_idx` (`Comanda` ASC) VISIBLE,
  INDEX `fk_BevandeEffettiva_Bevanda_idx` (`NomeBevanda` ASC) VISIBLE,
  CONSTRAINT `fk_BevandeEffettiva_Comanda`
    FOREIGN KEY (`Comanda`)
    REFERENCES `pizzeriaDB`.`Comande` (`Numero`)
    ON DELETE CASCADE
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_BevandeEffettiva_Bevanda`
    FOREIGN KEY (`NomeBevanda`)
    REFERENCES `pizzeriaDB`.`Bevande` (`Nome`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Clienti`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Clienti` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Clienti` (
  `Nome` VARCHAR(45) NOT NULL,
  `Cognome` VARCHAR(45) NOT NULL,
  `Tavolo` INT UNSIGNED NOT NULL,
  `NumeroCommensali` INT UNSIGNED NOT NULL,
  PRIMARY KEY (`Nome`, `Cognome`, `Tavolo`),
  UNIQUE INDEX `Tavolo_UNIQUE` (`Tavolo` ASC) VISIBLE,
  CONSTRAINT `fk_Clienti`
    FOREIGN KEY (`Tavolo`)
    REFERENCES `pizzeriaDB`.`Tavoli` (`NumeroTavolo`)
```

```
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Turni_Tavoli`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Turni_Tavoli` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Turni_Tavoli` (
  `Tavolo` INT UNSIGNED NOT NULL,
  `Turno` VARCHAR(45) NOT NULL,
  `Data` DATE NOT NULL,
  PRIMARY KEY (`Tavolo`, `Data`, `Turno`),
  INDEX `fk_TurniT_Turno_idx` (`Turno` ASC) VISIBLE,
  INDEX `fk_TurniT_Tavolo_idx` (`Tavolo` ASC) VISIBLE,
  CONSTRAINT `fk_TurniT_Tavolo`
    FOREIGN KEY (`Tavolo`)
    REFERENCES `pizzeriaDB`.`Tavoli` (`NumeroTavolo`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_TurniT_Turno`
    FOREIGN KEY (`Turno`)
    REFERENCES `pizzeriaDB`.`Turni` (`Nome`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Turni_Camerieri`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Turni_Camerieri` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Turni_Camerieri` (
  `NomeC` VARCHAR(45) NOT NULL,
  `CognomeC` VARCHAR(45) NOT NULL,
  `Turno` VARCHAR(45) NOT NULL,
  `Data` DATE NOT NULL,
  PRIMARY KEY (`NomeC`, `CognomeC`, `Data`),
  INDEX `fk_TurniC_Turno_idx` (`Turno` ASC) VISIBLE,
```

```
  INDEX `fk_TurniC_Cameriere_idx` (`NomeC` ASC, `CognomeC` ASC) VISIBLE,
  CONSTRAINT `fk_TurniC_Cameriere`
    FOREIGN KEY (`NomeC` , `CognomeC`)
    REFERENCES `pizzeriaDB`.`Camerieri` (`Nome` , `Cognome`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_TurniC_Turno`
    FOREIGN KEY (`Turno`)
    REFERENCES `pizzeriaDB`.`Turni` (`Nome`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Condimenti`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Condimenti` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Condimenti` (
  `Pizza` VARCHAR(45) NOT NULL,
  `Ingrediente` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`Pizza`, `Ingrediente`),
  INDEX `fk_Condimenti_Ingrediente_idx` (`Ingrediente` ASC) VISIBLE,
  INDEX `fk_Condimenti_Pizza_idx` (`Pizza` ASC) VISIBLE,
  CONSTRAINT `fk_Condimenti_Ingrediente`
    FOREIGN KEY (`Ingrediente`)
    REFERENCES `pizzeriaDB`.`Ingredienti` (`Nome`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_Condimenti_Pizza`
    FOREIGN KEY (`Pizza`)
    REFERENCES `pizzeriaDB`.`Pizze` (`Nome`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

```
-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Aggiunte`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Aggiunte` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Aggiunte` (
  `IdPizza` INT UNSIGNED NOT NULL,
  `Comanda` INT UNSIGNED NOT NULL,
  `Ingrediente` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`IdPizza`, `Comanda`, `Ingrediente`),
  INDEX `fk_Aggiunta_Ingrediente_idx` (`Ingrediente` ASC) VISIBLE,
  INDEX `fk_Aggiunta_IdPizza_idx` (`IdPizza` ASC, `Comanda` ASC) VISIBLE,
  CONSTRAINT `fk_Aggiunta_IdPizza`
    FOREIGN KEY (`IdPizza` , `Comanda`)
    REFERENCES `pizzeriaDB`.`Pizze_effettive` (`Id` , `Comanda`)
    ON DELETE CASCADE
    ON UPDATE NO ACTION,
  CONSTRAINT `fk_Aggiunta_Ingrediente`
    FOREIGN KEY (`Ingrediente`)
    REFERENCES `pizzeriaDB`.`Ingredienti` (`Nome`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table `pizzeriaDB`.`Utenti`
-- -----------------------------------------------------
DROP TABLE IF EXISTS `pizzeriaDB`.`Utenti` ;
CREATE TABLE IF NOT EXISTS `pizzeriaDB`.`Utenti` (
  `Username` VARCHAR(91) NOT NULL,
  `Password` CHAR(32) NOT NULL,
  `Ruolo` ENUM('Manager', 'Cameriere', 'Pizzaiolo', 'Barista') NOT NULL,
  PRIMARY KEY (`Username`))
ENGINE = InnoDB;
```

```
-- -----------------------------------------------------
-- Users and privileges
-- -----------------------------------------------------
SET SQL_MODE = '';
DROP USER IF EXISTS Login;
SET
SQL_MODE='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO
_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION';
CREATE USER 'Login' IDENTIFIED BY 'Login';
GRANT EXECUTE ON procedure `pizzeriaDB`.`login` TO 'Login';

SET SQL_MODE = '';
DROP USER IF EXISTS Manager;
SET
SQL_MODE='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE ,
NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION';
CREATE USER 'Manager' IDENTIFIED BY 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`registra_cliente` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`crea_turno` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`crea_turno_cameriere` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`crea_turno_tavolo` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`inserisci_pizza_menu` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`inserisci_ingrediente_menu` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`inserisci_condimento_pizza` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`inserisci_bevanda_menu` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`aggiorna_quantita_ingrediente` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`aggiorna_quantita_bevanda` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`stampa_scontrino` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`visualizza_entrata_giornaliera` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`visualizza_entrata_mensile` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`associa_cameriere_tavolo` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`togli_cameriere_tavolo` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`registra_pagamento_scontrino` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`inserisci_cameriere` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`inserisci_tavolo` TO 'Manager';
GRANT EXECUTE ON procedure `pizzeriaDB`.`cancella_turno_cameriere` TO 'Manager';
```

```
SET SQL_MODE = '';
DROP USER IF EXISTS Pizzaiolo;
SET
SQL_MODE='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO
_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION';
CREATE USER 'Pizzaiolo' IDENTIFIED BY 'Pizzaiolo';
GRANT EXECUTE ON procedure `pizzeriaDB`.`visualizza_pizze_da_preparare` TO 'Pizzaiolo';
GRANT EXECUTE ON procedure `pizzeriaDB`.`segna_pizza_pronta` TO 'Pizzaiolo';

SET SQL_MODE = '';
DROP USER IF EXISTS Barista;
SET
SQL_MODE='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO
_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION';
CREATE USER 'Barista' IDENTIFIED BY 'Barista';
GRANT EXECUTE ON procedure `pizzeriaDB`.`visualizza_bevande_da_preparare` TO 'Barista';
GRANT EXECUTE ON procedure `pizzeriaDB`.`segna_bevanda_pronta` TO 'Barista';

SET SQL_MODE = '';
DROP USER IF EXISTS Cameriere;
SET
SQL_MODE='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO
_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION';
CREATE USER 'Cameriere' IDENTIFIED BY 'Cameriere';
GRANT EXECUTE ON procedure `pizzeriaDB`.`visualizza_tavoli_occupati` TO 'Cameriere';
GRANT EXECUTE ON procedure `pizzeriaDB`.`visualizza_tavoli_comande_servite` TO
'Cameriere';
GRANT EXECUTE ON procedure `pizzeriaDB`.`registra_comanda` TO 'Cameriere';
GRANT EXECUTE ON procedure `pizzeriaDB`.`visualizza_cosa_e_dove_servire` TO
'Cameriere';
GRANT EXECUTE ON procedure `pizzeriaDB`.`ordina_pizza` TO 'Cameriere';
GRANT EXECUTE ON procedure `pizzeriaDB`.`inserisci_aggiunta_pizza` TO 'Cameriere';
GRANT EXECUTE ON procedure `pizzeriaDB`.`ordina_bevanda` TO 'Cameriere';
GRANT EXECUTE ON procedure `pizzeriaDB`.`segna_pizza_ordinata` TO 'Cameriere';
GRANT EXECUTE ON procedure `pizzeriaDB`.`segna_pizza_servita` TO 'Cameriere';
GRANT EXECUTE ON procedure `pizzeriaDB`.`segna_bevanda_servita` TO 'Cameriere';
```

```
SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;


-- -----------------------------------------------------
-- Utenti
-- -----------------------------------------------------

-- password di Elisa_Moltisanti: Elisa

-- password di Federica_Bellina: Federica

-- password di Alessandro_Cobisi: Alessandro

-- password di Francesca_Pavone: Francesca

-- password di Giorgio_Lentini: Giorgio
```

Codice del Front-End

main.c

```
#include <stdio.h>
#include "view/login.h"
#include "controller/login.h"
#include "model/db.h"
#include "utils/dotenv.h"
#include "utils/io.h"
#include "utils/validation.h"

#define check_env_failing(varname)                                      \
if(getenv((varname)) == NULL) {                                         \
        fprintf(stderr, "[FATAL] env variable %s not set\n", (varname));    \
        ret = false;                                                    \
}
static bool validate_dotenv(void)
{
        bool ret = true;

        check_env_failing("HOST");
        check_env_failing("DB");
        check_env_failing("LOGIN_USER");
        check_env_failing("LOGIN_PASS");
        check_env_failing("MANAGER_USER");
        check_env_failing("MANAGER_PASS");
        check_env_failing("CAMERIERE_USER");
        check_env_failing("CAMERIERE_PASS");
        check_env_failing("PIZZAIOLO_USER");
        check_env_failing("PIZZAIOLO_PASS");
        check_env_failing("BARISTA_USER");
        check_env_failing("BARISTA_PASS");

        return ret;
}
#undef set_env_failing

int main()
{
        if(env_load(".", false) != 0)
                return 1;
        if(!validate_dotenv())
                return 1;
        if(!init_validation())
```

```
            return 1;
        if(!init_db())
            return 1;

        if(initialize_io()) {
            do {
                if(!login())
                    fprintf(stderr, "Impossibile effettuare il login\n");
                db_switch_to_login();
            } while(ask_for_relogin());
        }
        fini_db();
        fini_validation();
        puts("\nAlla prossima!");
        return 0;
}
```

.env

```
HOST=127.0.0.1
DB=pizzeriaDB
PORT=3306

LOGIN_USER=Login
LOGIN_PASS=Login
MANAGER_USER=Manager
MANAGER_PASS=Manager
CAMERIERE_USER=Cameriere
CAMERIERE_PASS=Cameriere
PIZZAIOLO_USER=Pizzaiolo
PIZZAIOLO_PASS=Pizzaiolo
BARISTA_USER=Barista
BARISTA_PASS=Barista
```

MODEL

db.h

```
#pragma once
#include <stdbool.h>
#include <stdlib.h>

extern bool init_db(void);
extern void fini_db(void);

#define DATE_LEN 11
#define TIME_LEN 3
#define DATETIME_LEN 20

#define USERNAME_LEN 91
#define PASSWORD_LEN 45
struct credentials {
        char username[USERNAME_LEN];
        char password[PASSWORD_LEN];
};

typedef enum {
        LOGIN_ROLE,
        MANAGER,
        CAMERIERE,
        PIZZAIOLO,
        BARISTA,
        FAILED_LOGIN
} role_t;

extern void db_switch_to_login(void);
extern role_t attempt_login(struct credentials *cred);

extern void db_switch_to_manager(void);
extern void db_switch_to_cameriere(void);
extern void db_switch_to_pizzaiolo(void);
extern void db_switch_to_barista(void);


#define NOME_TURNO_LEN 45
struct turno {
   char nome[NOME_TURNO_LEN];
   char oraInizio[TIME_LEN];
```

```
    char oraFine[TIME_LEN];
};
extern void do_crea_turno(struct turno *turno);


#define NOME_CAMERIERE_LEN 45
#define COGNOME_CAMERIERE_LEN 45
struct turno_cameriere {
    char nomeC[NOME_CAMERIERE_LEN];
    char cognomeC[COGNOME_CAMERIERE_LEN];
    char turno[NOME_TURNO_LEN];
    char data[DATE_LEN];
};
extern bool do_crea_turno_cameriere(struct turno_cameriere *turnoCameriere);
extern void do_cancella_turno_cameriere(struct turno_cameriere *turnoCameriere);


struct turno_tavolo {
    unsigned int tavolo;
    char turno[NOME_TURNO_LEN];
    char data[DATE_LEN];
};
extern bool do_crea_turno_tavolo(struct turno_tavolo *turnoTavolo);


struct cameriere_tavolo {
    char nomeC[NOME_CAMERIERE_LEN];
    char cognomeC[COGNOME_CAMERIERE_LEN];
    unsigned int tavolo;
};
extern bool do_associa_cameriere_tavolo(struct cameriere_tavolo *camerieretavolo);
extern void do_togli_cameriere_tavolo(struct cameriere_tavolo *camerieretavolo);


#define NOME_CLIENTE_LEN 45
#define COGNOME_CLIENTE_LEN 45
struct cliente {
    char nome[NOME_CLIENTE_LEN];
    char cognome[COGNOME_CLIENTE_LEN];
    unsigned int numeroCommensali;
};
extern int do_registra_cliente(struct cliente *cliente);
```

```
#define NOME_PRODOTTO_LEN 45
#define PREZZO_PIZZA_LEN 6
struct pizza {
   char nome[NOME_PRODOTTO_LEN];
   char prezzo[PREZZO_PIZZA_LEN];
};
extern void do_inserisci_pizza_menu(struct pizza *pizza);


#define PREZZO_INGREDIENTE_LEN 5
struct ingrediente {
   char nome[NOME_PRODOTTO_LEN];
   char prezzo[PREZZO_INGREDIENTE_LEN];
};
extern void do_inserisci_ingrediente_menu(struct ingrediente *ingrediente);


struct condimento {
   char pizza[NOME_PRODOTTO_LEN];
   char ingrediente[NOME_PRODOTTO_LEN];
};
extern bool do_inserisci_condimento_pizza(struct condimento *condimento);


#define PREZZO_BEVANDA_LEN 7
struct bevanda {
   char nome[NOME_PRODOTTO_LEN];
   char prezzo[PREZZO_BEVANDA_LEN];
};
extern void do_inserisci_bevanda_menu(struct bevanda *bevanda);


struct prodotto_quantita {
   char nome[NOME_PRODOTTO_LEN];
   unsigned int quantita;
};
extern void do_aggiorna_quantita_ingrediente(struct prodotto_quantita *ingredienteQuantita);
extern void do_aggiorna_quantita_bevanda(struct prodotto_quantita *bevandaQuantita);


struct tavolo {
  unsigned int numeroTavolo;
};
extern int do_registra_comanda(struct tavolo *tavolo, char *username);
```

```
#define PREZZO_SCONTRINO_LEN 8
struct scontrino {
   char prezzo[PREZZO_SCONTRINO_LEN];
   char id[DATETIME_LEN];
};
extern struct scontrino *do_stampa_scontrino(struct tavolo *tavolo);
extern void do_registra_pagamento_scontrino(struct scontrino *scontrino);


#define ENTRATE_LEN 10
struct entrata {
   char tot[ENTRATE_LEN];
};
struct anno_mese {
   short int anno;
   unsigned int mese;
};
extern struct entrata *do_visualizza_entrata_mensile(struct anno_mese *annoMese);
extern struct entrata *do_visualizza_entrata_giornaliera(char giorno[DATE_LEN]);


struct  nuovo_tavolo {
   unsigned int numeroTavolo;
   unsigned int numeroPosti;
};
extern void do_inserisci_tavolo(struct nuovo_tavolo *nuovoTavolo);


struct nuovo_cameriere {
   char nome[NOME_CAMERIERE_LEN];
   char cognome[COGNOME_CAMERIERE_LEN];
};
extern void do_inserisci_cameriere(struct nuovo_cameriere *nuovoCameriere);


struct lista_tavoli {
   unsigned int num_entries;
   struct tavolo tavolo[];
};
extern struct lista_tavoli *do_visualizza_tavoli_occupati(char *username);
extern struct lista_tavoli *do_visualizza_tavoli_comande_servite(char *username);
```

```
struct prodotto_da_servire {
    unsigned int id;
    char nome[NOME_PRODOTTO_LEN];
};
struct comande_da_servire {
    unsigned int comanda;
    unsigned int tavolo;
    size_t num_prodotti;
    struct prodotto_da_servire *prodottiDaServire;
};
struct da_servire {
    size_t num_comande;
    struct comande_da_servire comandeDaServire[];
};
extern struct da_servire *do_visualizza_cosa_e_dove_servire(char *username);
extern void free_da_servire(struct da_servire *daServire);


struct bevanda_effettiva {
    char nomeBevanda[NOME_PRODOTTO_LEN];
    unsigned int comanda;
};
extern bool do_ordina_bevanda(struct bevanda_effettiva *bevandaEffettiva, char *username);


struct bevanda_stato {
    unsigned int idBevanda;
    unsigned int comanda;
};
extern bool do_segna_bevanda_servita(struct bevanda_stato *bevandaServita, char *username);
extern bool do_segna_bevanda_pronta(struct bevanda_stato *bevandaPronta);


struct pizza_effettiva {
    char nomePizza[NOME_PRODOTTO_LEN];
    unsigned int comanda;
};
extern int do_ordina_pizza(struct pizza_effettiva *pizzaEffettiva, char *username);


struct aggiunta_pizza {
    unsigned int idPizza;
    unsigned int comanda;
```

```
    char ingrediente[NOME_PRODOTTO_LEN];
};
extern void do_inserisci_aggiunta_pizza(struct aggiunta_pizza *aggiunta);


struct pizza_stato {
    unsigned int idPizza;
    unsigned int comanda;
};
extern void do_segna_pizza_ordinata(struct pizza_stato *pizzaOrdinata);
extern bool do_segna_pizza_servita(struct pizza_stato *pizzaServita, char *username);
extern bool do_segna_pizza_pronta(struct pizza_stato *pizzaPronta);


struct aggiunta {
    char ingrediente[NOME_PRODOTTO_LEN];
};
struct pizza_info {
    unsigned int comanda;
    unsigned int idPizza;
    char nomePizza[NOME_PRODOTTO_LEN];
    size_t num_aggiunte;
    struct aggiunta *aggiunte;
};
struct pizze_da_preparare {
    size_t num_pizze;
    struct pizza_info pizzaInfo[];
};
extern struct pizze_da_preparare *do_visualizza_pizze_da_preparare(void);
extern void free_pizze_da_preparare(struct pizze_da_preparare *pizzeDaPreparare);


struct bevanda_da_preparare {
    unsigned int comanda;
    unsigned int idBevanda;
    char nomeBevanda[NOME_PRODOTTO_LEN];
};
struct lista_bevande_da_preparare {
    unsigned int num_entries;
    struct bevanda_da_preparare bevandaDaPreparare[];
};
extern struct lista_bevande_da_preparare *do_visualizza_bevande_da_preparare(void);
```

db.c

```
#include <stdlib.h>
```

```
#include <stdio.h>
#include <string.h>
#include <mysql.h>
#include <assert.h>

#include "db.h"
#include "../utils/db.h"

static MYSQL *conn;

static MYSQL_STMT *login_procedure;

//MANAGER
static MYSQL_STMT *crea_turno;
static MYSQL_STMT *crea_turno_cameriere;
static MYSQL_STMT *crea_turno_tavolo;
static MYSQL_STMT *associa_cameriere_tavolo;
static MYSQL_STMT *togli_cameriere_tavolo;
static MYSQL_STMT *registra_cliente;
static MYSQL_STMT *inserisci_pizza_menu;
static MYSQL_STMT *inserisci_ingrediente_menu;
static MYSQL_STMT *inserisci_condimento_pizza;
static MYSQL_STMT *inserisci_bevanda_menu;
static MYSQL_STMT *aggiorna_quantita_ingrediente;
static MYSQL_STMT *aggiorna_quantita_bevanda;
static MYSQL_STMT *stampa_scontrino;
static MYSQL_STMT *registra_pagamento_scontrino;
static MYSQL_STMT *visualizza_entrata_giornaliera;
static MYSQL_STMT *visualizza_entrata_mensile;
static MYSQL_STMT *inserisci_tavolo;
static MYSQL_STMT *inserisci_cameriere;
static MYSQL_STMT *cancella_turno_cameriere;

//CAMERIERE
static MYSQL_STMT *visualizza_tavoli_occupati;
static MYSQL_STMT *visualizza_tavoli_comande_servite;
static MYSQL_STMT *registra_comanda;
static MYSQL_STMT *visualizza_cosa_e_dove_servire;
static MYSQL_STMT *ordina_pizza;
static MYSQL_STMT *inserisci_aggiunta_pizza;
static MYSQL_STMT *segna_pizza_ordinata;
static MYSQL_STMT *ordina_bevanda;
static MYSQL_STMT *segna_pizza_servita;
static MYSQL_STMT *segna_bevanda_servita;
```

```
//PIZZAIOLO
static MYSQL_STMT *visualizza_pizze_da_preparare;
static MYSQL_STMT *segna_pizza_pronta;

//BARISTA
static MYSQL_STMT *visualizza_bevande_da_preparare;
static MYSQL_STMT *segna_bevanda_pronta;


static void close_prepared_stmts(void)
{
        if(login_procedure) {
                mysql_stmt_close(login_procedure);
                login_procedure = NULL;
        }

   //MANAGER
   if(crea_turno) {
      mysql_stmt_close(crea_turno);
      crea_turno = NULL;
   }
   if(crea_turno_cameriere) {
      mysql_stmt_close(crea_turno_cameriere);
      crea_turno_cameriere = NULL;
   }
   if(crea_turno_tavolo) {
      mysql_stmt_close(crea_turno_tavolo);
      crea_turno_tavolo = NULL;
   }
   if(associa_cameriere_tavolo) {
      mysql_stmt_close(associa_cameriere_tavolo);
      associa_cameriere_tavolo = NULL;
   }
   if(togli_cameriere_tavolo) {
      mysql_stmt_close(togli_cameriere_tavolo);
      togli_cameriere_tavolo = NULL;
   }
   if(registra_cliente) {
      mysql_stmt_close(registra_cliente);
      registra_cliente = NULL;
   }
   if(inserisci_pizza_menu) {
      mysql_stmt_close(inserisci_pizza_menu);
```

```
    inserisci_pizza_menu = NULL;
}
if(inserisci_ingrediente_menu) {
    mysql_stmt_close(inserisci_ingrediente_menu);
    inserisci_ingrediente_menu = NULL;
}
if(inserisci_condimento_pizza) {
    mysql_stmt_close(inserisci_condimento_pizza);
    inserisci_condimento_pizza = NULL;
}
if(inserisci_bevanda_menu) {
    mysql_stmt_close(inserisci_bevanda_menu);
    inserisci_bevanda_menu = NULL;
}
if(aggiorna_quantita_ingrediente) {
    mysql_stmt_close(aggiorna_quantita_ingrediente);
    aggiorna_quantita_ingrediente = NULL;
}
if(aggiorna_quantita_bevanda) {
    mysql_stmt_close(aggiorna_quantita_bevanda);
    aggiorna_quantita_bevanda = NULL;
}
if(stampa_scontrino) {
    mysql_stmt_close(stampa_scontrino);
    stampa_scontrino = NULL;
}
if(registra_pagamento_scontrino) {
    mysql_stmt_close(registra_pagamento_scontrino);
    registra_pagamento_scontrino = NULL;
}
if(visualizza_entrata_giornaliera) {
    mysql_stmt_close(visualizza_entrata_giornaliera);
    visualizza_entrata_giornaliera = NULL;
}
if(visualizza_entrata_mensile) {
    mysql_stmt_close(visualizza_entrata_mensile);
    visualizza_entrata_mensile = NULL;
}
if(inserisci_tavolo) {
    mysql_stmt_close(inserisci_tavolo);
    inserisci_tavolo = NULL;
}
if(inserisci_cameriere) {
    mysql_stmt_close(inserisci_cameriere);
```

```
    inserisci_cameriere = NULL;
}
if(cancella_turno_cameriere) {
    mysql_stmt_close(cancella_turno_cameriere);
    cancella_turno_cameriere = NULL;
}

//CAMERIERE
if(visualizza_tavoli_occupati) {
    mysql_stmt_close(visualizza_tavoli_occupati);
    visualizza_tavoli_occupati = NULL;
}
if(visualizza_tavoli_comande_servite) {
    mysql_stmt_close(visualizza_tavoli_comande_servite);
    visualizza_tavoli_comande_servite = NULL;
}
if(registra_comanda) {
    mysql_stmt_close(registra_comanda);
    registra_comanda = NULL;
}
if(visualizza_cosa_e_dove_servire) {
    mysql_stmt_close(visualizza_cosa_e_dove_servire);
    visualizza_cosa_e_dove_servire = NULL;
}
if(ordina_pizza) {
    mysql_stmt_close(ordina_pizza);
    ordina_pizza = NULL;
}
if(inserisci_aggiunta_pizza) {
    mysql_stmt_close(inserisci_aggiunta_pizza);
    inserisci_aggiunta_pizza = NULL;
}
if(segna_pizza_ordinata) {
    mysql_stmt_close(segna_pizza_ordinata);
    segna_pizza_ordinata = NULL;
}
if(ordina_bevanda) {
    mysql_stmt_close(ordina_bevanda);
    ordina_bevanda = NULL;
}
if(segna_pizza_servita) {
    mysql_stmt_close(segna_pizza_servita);
    segna_pizza_servita = NULL;
}
```

```
    if(segna_bevanda_servita) {
        mysql_stmt_close(segna_bevanda_servita);
        segna_bevanda_servita = NULL;
    }

    //PIZZAIOLO
    if(visualizza_pizze_da_preparare) {
        mysql_stmt_close(visualizza_pizze_da_preparare);
        visualizza_pizze_da_preparare = NULL;
    }
    if(segna_pizza_pronta) {
        mysql_stmt_close(segna_pizza_pronta);
        segna_pizza_pronta = NULL;
    }

    //BARISTA
    if(visualizza_bevande_da_preparare) {
        mysql_stmt_close(visualizza_bevande_da_preparare);
        visualizza_bevande_da_preparare = NULL;
    }
    if(segna_bevanda_pronta) {
        mysql_stmt_close(segna_bevanda_pronta);
        segna_bevanda_pronta = NULL;
    }
}
static bool initialize_prepared_stmts(role_t for_role)
{
        switch(for_role) {

                case LOGIN_ROLE:
                        if(!setup_prepared_stmt(&login_procedure, "call login(?, ?, ?)", conn)) {
                                print_stmt_error(login_procedure, "Impossibile inizilaizzare lo
statement di login\n");
                                return false;
                        }
                        break;

                case MANAGER:
                        if(!setup_prepared_stmt(&crea_turno, "call crea_turno(?, ?, ?)", conn)) {
                                print_stmt_error(crea_turno, "Impossibile inizilaizzare lo statement di
creazione di un turno\n");
                                return false;
                        }
        if(!setup_prepared_stmt(&crea_turno_cameriere, "call crea_turno_cameriere(?, ?, ?, ?)",
```

```
conn)) {
            print_stmt_error(crea_turno_cameriere, "Impossibile inizilaizzare lo statement di creazione
del turno di un cameriere\n");
            return false;
        }
        if(!setup_prepared_stmt(&crea_turno_tavolo, "call crea_turno_tavolo(?, ?, ?)", conn)) {
            print_stmt_error(crea_turno_tavolo, "Impossibile inizilaizzare lo statement di creazione del
turno di un tavolo\n");
            return false;
        }
        if(!setup_prepared_stmt(&associa_cameriere_tavolo, "call associa_cameriere_tavolo(?, ?, ?)",
conn)) {
            print_stmt_error(associa_cameriere_tavolo, "Impossibile inizilaizzare lo statement di
associazione di un cameriere ad un tavolo\n");
            return false;
        }
        if(!setup_prepared_stmt(&togli_cameriere_tavolo, "call togli_cameriere_tavolo(?, ?, ?)",
conn)) {
            print_stmt_error(togli_cameriere_tavolo, "Impossibile inizilaizzare lo statement di
rimozione di un tavolo ad un cameriere\n");
            return false;
        }
        if(!setup_prepared_stmt(&registra_cliente, "call registra_cliente(?, ?, ?, ?)", conn)) {
            print_stmt_error(registra_cliente, "Impossibile inizilaizzare lo statement di registrazione di
un cliente\n");
            return false;
        }
        if(!setup_prepared_stmt(&inserisci_pizza_menu, "call inserisci_pizza_menu(?, ?)", conn)) {
            print_stmt_error(inserisci_pizza_menu, "Impossibile inizilaizzare lo statement di
inserimento di una pizza nel menu\n");
            return false;
        }
        if(!setup_prepared_stmt(&inserisci_ingrediente_menu, "call
inserisci_ingrediente_menu(?, ?)", conn)) {
            print_stmt_error(inserisci_ingrediente_menu, "Impossibile inizilaizzare lo statement di
inserimento di un ingrediente nel menu\n");
            return false;
        }
        if(!setup_prepared_stmt(&inserisci_condimento_pizza, "call
inserisci_condimento_pizza(?, ?)", conn)) {
            print_stmt_error(inserisci_condimento_pizza, "Impossibile inizilaizzare lo statement di
inserimento di un condimento di una pizza\n");
            return false;
        }
```

```
        if(!setup_prepared_stmt(&inserisci_bevanda_menu, "call inserisci_bevanda_menu(?, ?)",
conn)) {
            print_stmt_error(inserisci_bevanda_menu, "Impossibile inizilaizzare lo statement di
inserimento di una bevanda nel menu\n");
            return false;
        }
        if(!setup_prepared_stmt(&aggiorna_quantita_ingrediente, "call
aggiorna_quantita_ingrediente(?, ?)", conn)) {
            print_stmt_error(aggiorna_quantita_ingrediente, "Impossibile inizilaizzare lo statement di
aggiornamneto della quantita' di un ingrediente\n");
            return false;
        }
        if(!setup_prepared_stmt(&aggiorna_quantita_bevanda, "call
aggiorna_quantita_bevanda(?, ?)", conn)) {
            print_stmt_error(aggiorna_quantita_bevanda, "Impossibile inizilaizzare lo statement di
aggiornamneto della quantita' di una bevanda\n");
            return false;
        }
        if(!setup_prepared_stmt(&stampa_scontrino, "call stampa_scontrino(?, ?, ?)", conn)) {
            print_stmt_error(stampa_scontrino, "Impossibile inizilaizzare lo statement di stampa di
uno scontrino\n");
            return false;
        }
        if(!setup_prepared_stmt(&registra_pagamento_scontrino, "call
registra_pagamento_scontrino(?)", conn)) {
            print_stmt_error(registra_pagamento_scontrino, "Impossibile inizilaizzare lo statement di
registrazione del pagamento di uno scontrino\n");
            return false;
        }
        if(!setup_prepared_stmt(&visualizza_entrata_giornaliera, "call
visualizza_entrata_giornaliera(?, ?)", conn)) {
            print_stmt_error(visualizza_entrata_giornaliera, "Impossibile inizilaizzare lo statement di
visualizzazione delle entrate di una giornata\n");
            return false;
        }
        if(!setup_prepared_stmt(&visualizza_entrata_mensile, "call
visualizza_entrata_mensile(?, ?, ?)", conn)) {
            print_stmt_error(visualizza_entrata_mensile, "Impossibile inizilaizzare lo statement di
visualizzazione delle entrate di un mese\n");
            return false;
        }
        if(!setup_prepared_stmt(&inserisci_tavolo, "call inserisci_tavolo(?, ?)", conn)) {
            print_stmt_error(inserisci_tavolo, "Impossibile inizilaizzare lo statement di inserimento di
un nuovo tavolo\n");
```

```
            return false;
        }
        if(!setup_prepared_stmt(&inserisci_cameriere, "call inserisci_cameriere(?, ?)", conn)) {
            print_stmt_error(inserisci_cameriere, "Impossibile inizilaizzare lo statement di inserimento
di un nuovo cameriere\n");
            return false;
        }
        if(!setup_prepared_stmt(&cancella_turno_cameriere, "call
cancella_turno_cameriere(?, ?, ?, ?)", conn)) {
            print_stmt_error(cancella_turno_cameriere, "Impossibile inizilaizzare lo statement di
inserimento di un nuovo cameriere\n");
            return false;
        }
                    break;

                case CAMERIERE:
                    if(!setup_prepared_stmt(&visualizza_tavoli_occupati, "call
visualizza_tavoli_occupati(?)", conn)) {
                        print_stmt_error(visualizza_tavoli_occupati, "Impossibile inizilaizzare
lo statement di visualizzazione dei tavoli occupati\n");
                        return false;
                    }
        if(!setup_prepared_stmt(&visualizza_tavoli_comande_servite, "call
visualizza_tavoli_comande_servite(?)", conn)) {
            print_stmt_error(visualizza_tavoli_comande_servite, "Impossibile inizilaizzare lo
statement di visualizzazione dei tavoli in cui le comande sono state completamente servite\n");
            return false;
        }
        if(!setup_prepared_stmt(&registra_comanda, "call registra_comanda(?, ?, ?)", conn)) {
            print_stmt_error(registra_comanda, "Impossibile inizilaizzare lo statement di registrazione
di una comanda\n");
            return false;
        }
        if(!setup_prepared_stmt(&visualizza_cosa_e_dove_servire, "call
visualizza_cosa_e_dove_servire(?)", conn)) {
            print_stmt_error(visualizza_cosa_e_dove_servire, "Impossibile inizilaizzare lo statement di
visualizzazione di cosa servire e in che tavolo\n");
            return false;
        }
        if(!setup_prepared_stmt(&ordina_pizza, "call ordina_pizza(?, ?, ?, ?)", conn)) {
            print_stmt_error(ordina_pizza, "Impossibile inizilaizzare lo statement di ordinazione di una
pizza\n");
            return false;
        }
```

```
        if(!setup_prepared_stmt(&inserisci_aggiunta_pizza, "call inserisci_aggiunta_pizza(?, ?, ?)",
conn)) {
            print_stmt_error(inserisci_aggiunta_pizza, "Impossibile inizilaizzare lo statement di
inseirimento di un'aggiunta_pizza ad una pizza\n");
            return false;
        }
        if(!setup_prepared_stmt(&segna_pizza_ordinata, "call segna_pizza_ordinata(?, ?)", conn)) {
            print_stmt_error(segna_pizza_ordinata, "Impossibile inizilaizzare lo statement per segnare
una pizza come ordinata\n");
            return false;
        }
        if(!setup_prepared_stmt(&ordina_bevanda, "call ordina_bevanda(?, ?, ?)", conn)) {
            print_stmt_error(ordina_bevanda, "Impossibile inizilaizzare lo statement di ordinazione di
una bevanda\n");
            return false;
        }
        if(!setup_prepared_stmt(&segna_pizza_servita, "call segna_pizza_servita(?, ?, ?)", conn)) {
            print_stmt_error(segna_pizza_servita, "Impossibile inizilaizzare lo statement per segnare
una pizza come servita\n");
            return false;
        }
        if(!setup_prepared_stmt(&segna_bevanda_servita, "call segna_bevanda_servita(?, ?, ?)",
conn)) {
            print_stmt_error(segna_bevanda_servita, "Impossibile inizilaizzare lo statement per
segnare una bevanda come servita\n");
            return false;
        }
        break;

    case PIZZAIOLO:
        if(!setup_prepared_stmt(&visualizza_pizze_da_preparare, "call
visualizza_pizze_da_preparare()", conn)) {
            print_stmt_error(visualizza_pizze_da_preparare, "Impossibile inizilaizzare lo statement di
visualizzazione delle pizze da preparare\n");
            return false;
        }
        if(!setup_prepared_stmt(&segna_pizza_pronta, "call segna_pizza_pronta(?, ?)", conn)) {
            print_stmt_error(segna_pizza_pronta, "Impossibile inizilaizzare lo statement per segnare
una pizza come pronta\n");
            return false;
        }
        break;

    case BARISTA:
```

```
        if(!setup_prepared_stmt(&visualizza_bevande_da_preparare, "call
visualizza_bevande_da_preparare()", conn)) {
            print_stmt_error(visualizza_bevande_da_preparare, "Impossibile inizilaizzare lo statement
di visualizzazione delle bevande da preparare\n");
            return false;
        }
        if(!setup_prepared_stmt(&segna_bevanda_pronta, "call segna_bevanda_pronta(?, ?)", conn))
{
            print_stmt_error(segna_bevanda_pronta, "Impossibile inizilaizzare lo statement per
segnare una bevanda come pronta\n");
            return false;
        }
        break;

                default:
                        fprintf(stderr, "[FATAL] Unexpected role to prepare statements.\n");
                        exit(EXIT_FAILURE);
        }

        return true;
}

bool init_db(void)
{
        unsigned int timeout = 300;
        bool reconnect = true;

        conn = mysql_init(NULL);
        if(conn == NULL) {
                finish_with_error(conn, "mysql_init() failed (probably out of memory)\n");
        }

        if(mysql_real_connect(conn, getenv("HOST"), getenv("LOGIN_USER"),
getenv("LOGIN_PASS"), getenv("DB"),
                                atoi(getenv("PORT")), NULL,
                                CLIENT_MULTI_STATEMENTS | CLIENT_MULTI_RESULTS |
CLIENT_COMPRESS | CLIENT_INTERACTIVE | CLIENT_REMEMBER_OPTIONS) ==
NULL) {
                finish_with_error(conn, "mysql_real_connect() failed\n");
        }

        if (mysql_options(conn, MYSQL_OPT_CONNECT_TIMEOUT, &timeout)) {
                print_error(conn, "[mysql_options] failed.");
        }
```

```
	if(mysql_options(conn, MYSQL_OPT_RECONNECT, &reconnect)) {
		print_error(conn, "[mysql_options] failed.");
	}
#ifndef NDEBUG
	mysql_debug("d:t:O,/tmp/client.trace");
	if(mysql_dump_debug_info(conn)) {
		print_error(conn, "[debug_info] failed.");
	}
#endif

	return initialize_prepared_stmts(LOGIN_ROLE);
}


void fini_db(void)
{
	close_prepared_stmts();

	mysql_close(conn);
}


role_t attempt_login(struct credentials *cred)
{
	MYSQL_BIND param[3]; // Used both for input and output
	int role = 0;

	// Prepare parameters
	set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, cred->username,
strlen(cred->username));
	set_binding_param(&param[1], MYSQL_TYPE_VAR_STRING, cred->password,
strlen(cred->password));
	set_binding_param(&param[2], MYSQL_TYPE_LONG, &role, sizeof(role));

	if(mysql_stmt_bind_param(login_procedure, param) != 0) { // Note _param
		print_stmt_error(login_procedure, "Could not bind parameters for login");
		role = FAILED_LOGIN;
		goto out;
	}

	// Run procedure
	if(mysql_stmt_execute(login_procedure) != 0) {
		print_stmt_error(login_procedure, "Could not execute login procedure");
		role = FAILED_LOGIN;
```

```
                goto out;
        }

        // Prepare output parameters
        set_binding_param(&param[0], MYSQL_TYPE_LONG, &role, sizeof(role));

        if(mysql_stmt_bind_result(login_procedure, param)) {
                print_stmt_error(login_procedure, "Could not retrieve output parameter");
                role = FAILED_LOGIN;
                goto out;
        }

        // Retrieve output parameter
        if(mysql_stmt_fetch(login_procedure)) {
                print_stmt_error(login_procedure, "Could not buffer results");
                role = FAILED_LOGIN;
                goto out;
        }

   out:
   mysql_stmt_free_result(login_procedure);
        while(mysql_stmt_next_result(login_procedure) != -1) {}
   mysql_stmt_reset(login_procedure);
        return role;
}


void db_switch_to_login(void)
{
        close_prepared_stmts();
        if(mysql_change_user(conn, getenv("LOGIN_USER"), getenv("LOGIN_PASS"),
getenv("DB"))) {
                fprintf(stderr, "mysql_change_user() failed: %s\n", mysql_error(conn));
                exit(EXIT_FAILURE);
        }
        if(!initialize_prepared_stmts(LOGIN_ROLE)) {
                fprintf(stderr, "[FATAL] Cannot initialize prepared statements.\n");
                exit(EXIT_FAILURE);
        }
}

void db_switch_to_manager(void)
{
   close_prepared_stmts();
```

```
    if(mysql_change_user(conn, getenv("MANAGER_USER"), getenv("MANAGER_PASS"),
getenv("DB"))) {
        fprintf(stderr, "mysql_change_user() failed: %s\n", mysql_error(conn));
        exit(EXIT_FAILURE);
    }
    if(!initialize_prepared_stmts(MANAGER)) {
        fprintf(stderr, "[FATAL] Cannot initialize prepared statements.\n");
        exit(EXIT_FAILURE);
    }
}

void db_switch_to_cameriere(void)
{
    close_prepared_stmts();
    if(mysql_change_user(conn, getenv("CAMERIERE_USER"), getenv("CAMERIERE_PASS"),
getenv("DB"))) {
        fprintf(stderr, "mysql_change_user() failed: %s\n", mysql_error(conn));
        exit(EXIT_FAILURE);
    }
    if(!initialize_prepared_stmts(CAMERIERE)) {
        fprintf(stderr, "[FATAL] Cannot initialize prepared statements.\n");
        exit(EXIT_FAILURE);
    }
}

void db_switch_to_pizzaiolo(void)
{
    close_prepared_stmts();
    if(mysql_change_user(conn, getenv("PIZZAIOLO_USER"), getenv("PIZZAIOLO_PASS"),
getenv("DB"))) {
        fprintf(stderr, "mysql_change_user() failed: %s\n", mysql_error(conn));
        exit(EXIT_FAILURE);
    }
    if(!initialize_prepared_stmts(PIZZAIOLO)) {
        fprintf(stderr, "[FATAL] Cannot initialize prepared statements.\n");
        exit(EXIT_FAILURE);
    }
}

void db_switch_to_barista(void)
{
    close_prepared_stmts();
    if(mysql_change_user(conn, getenv("BARISTA_USER"), getenv("BARISTA_PASS"),
getenv("DB"))) {
```

```
        fprintf(stderr, "mysql_change_user() failed: %s\n", mysql_error(conn));
        exit(EXIT_FAILURE);
    }
    if(!initialize_prepared_stmts(BARISTA)) {
        fprintf(stderr, "[FATAL] Cannot initialize prepared statements.\n");
        exit(EXIT_FAILURE);
    }
}

//------------------ MANAGER ------------------

void do_crea_turno(struct turno *turno)
{
    MYSQL_BIND param[3];
    MYSQL_TIME oraInizio;
    MYSQL_TIME oraFine;

    //conversione dei tipi TIME mysql
    time_to_mysql_time(turno->oraInizio, &oraInizio);
    time_to_mysql_time(turno->oraFine, &oraFine);

    // bind dei parametri
    set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, turno->nome, strlen(turno-
>nome));
    set_binding_param(&param[1], MYSQL_TYPE_TIME, &oraInizio, sizeof(oraInizio));
    set_binding_param(&param[2], MYSQL_TYPE_TIME, &oraFine, sizeof(oraFine));

    if(mysql_stmt_bind_param(crea_turno, param) != 0) {
        print_stmt_error(crea_turno, "Impossibile effettuare il binding dei parametri per
do_crea_turno");
        return;
    }

    //esecuzione procedura
    if(mysql_stmt_execute(crea_turno) != 0) {
        print_stmt_error(crea_turno, "Impossibile eseguire la procedura 'crea turno'");
        return;
    }
    puts("\nTurno creato corretamente");



    mysql_stmt_free_result(crea_turno);
    mysql_stmt_reset(crea_turno);
```

```
}


bool do_crea_turno_cameriere(struct turno_cameriere *turnoCameriere)
{
   MYSQL_BIND param[4];
   MYSQL_TIME data;

   //conversione tipo data
   date_to_mysql_time(turnoCameriere->data, &data);

   // bind dei parametri
   set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, turnoCameriere->nomeC,
strlen(turnoCameriere->nomeC));
   set_binding_param(&param[1], MYSQL_TYPE_VAR_STRING, turnoCameriere->cognomeC,
strlen(turnoCameriere->cognomeC));
   set_binding_param(&param[2], MYSQL_TYPE_VAR_STRING, turnoCameriere->turno,
strlen(turnoCameriere->turno));
   set_binding_param(&param[3], MYSQL_TYPE_DATE, &data, sizeof(data));

   if(mysql_stmt_bind_param(crea_turno_cameriere, param) != 0) {
      print_stmt_error(crea_turno_cameriere, "Impossibile effettuare il binding dei parametri per
do_crea_turno_cameriere");
      return false;
   }

   //esecuzione procedura
   if(mysql_stmt_execute(crea_turno_cameriere) != 0) {
      print_stmt_error(crea_turno_cameriere, "Impossibile eseguire la procedura
'crea_turno_cameriere'");
      return false;
   }
   printf("\nCreazione del turno di lavoro di %s %s eseguita correttamente\n", turnoCameriere-
>nomeC, turnoCameriere->cognomeC);


   mysql_stmt_free_result(crea_turno_cameriere);
   mysql_stmt_reset(crea_turno_cameriere);
   return true;
}

bool do_crea_turno_tavolo(struct turno_tavolo *turnoTavolo)
{
   MYSQL_BIND param[3];
```

```
    MYSQL_TIME data;

    date_to_mysql_time(turnoTavolo->data, &data);

    set_binding_param(&param[0], MYSQL_TYPE_LONG, &turnoTavolo->tavolo,
sizeof(turnoTavolo->tavolo));
    set_binding_param(&param[1], MYSQL_TYPE_VAR_STRING, turnoTavolo->turno,
strlen(turnoTavolo->turno));
    set_binding_param(&param[2], MYSQL_TYPE_DATE, &data, sizeof(data));

    if(mysql_stmt_bind_param(crea_turno_tavolo, param) != 0) {
        print_stmt_error(crea_turno_tavolo, "Impossibile effettuare il binding dei parametri per
do_crea_turno_tavolo");
        return false;
    }

    if (mysql_stmt_execute(crea_turno_tavolo) != 0) {
        print_stmt_error(crea_turno_tavolo, "Impossibile eseguire la procedura 'crea_turno_tavolo'");
        return false;
    }
    printf("\nCreazione del turno per il tavolo %u eseguita correttamente\n", turnoTavolo->tavolo);


    mysql_stmt_free_result(crea_turno_tavolo);
    mysql_stmt_reset(crea_turno_tavolo);
    return true;
}

bool do_associa_cameriere_tavolo(struct cameriere_tavolo *camerieraTavolo)
{
    MYSQL_BIND param[3];

    set_binding_param(&param[0], MYSQL_TYPE_LONG, &camerieraTavolo->tavolo,
sizeof(camerieraTavolo->tavolo));
    set_binding_param(&param[1], MYSQL_TYPE_VAR_STRING, camerieraTavolo->nomeC,
strlen(camerieraTavolo->nomeC));
    set_binding_param(&param[2], MYSQL_TYPE_VAR_STRING, camerieraTavolo->cognomeC,
strlen(camerieraTavolo->cognomeC));

    if(mysql_stmt_bind_param(associa_cameriere_tavolo, param) != 0) {
        print_stmt_error(associa_cameriere_tavolo, "Impossibile effettuare il binding dei parametri per
do_associa_cameriere_tavolo");
        return false;
    }
```

```
    if (mysql_stmt_execute(associa_cameriere_tavolo) != 0) {
        print_stmt_error(associa_cameriere_tavolo, "Impossibile eseguire la procedura
'associa_cameriere_tavolo'");
        return false;
    }
    printf("\nIl cameriere %s %s e' stato associato al tavolo %u\n", camerieTavolo->nomeC,
camerieTavolo->cognomeC, camerieTavolo->tavolo);

    mysql_stmt_free_result(associa_cameriere_tavolo);
    mysql_stmt_reset(associa_cameriere_tavolo);
    return true;
}

void do_togli_cameriere_tavolo(struct cameriere_tavolo *camerieTavolo)
{
    MYSQL_BIND param[3];

    set_binding_param(&param[0], MYSQL_TYPE_LONG, &camerieTavolo->tavolo,
sizeof(camerieTavolo->tavolo));
    set_binding_param(&param[1], MYSQL_TYPE_VAR_STRING, camerieTavolo->nomeC,
strlen(camerieTavolo->nomeC));
    set_binding_param(&param[2], MYSQL_TYPE_VAR_STRING, camerieTavolo->cognomeC,
strlen(camerieTavolo->cognomeC));

    if(mysql_stmt_bind_param(togli_cameriere_tavolo, param) != 0) {
        print_stmt_error(togli_cameriere_tavolo, "Impossibile effettuare il binding dei parametri per
do_togli_cameriere_tavolo");
        return;
    }

    if (mysql_stmt_execute(togli_cameriere_tavolo) != 0) {
        print_stmt_error(togli_cameriere_tavolo, "Impossibile eseguire la procedura
'togli_cameriere_tavolo'");
        return;
    }
    printf("\nAl cameriere %s %s e' stato tolto il tavolo %u\n", camerieTavolo->nomeC,
camerieTavolo->cognomeC, camerieTavolo->tavolo);

    mysql_stmt_free_result(togli_cameriere_tavolo);
    mysql_stmt_reset(togli_cameriere_tavolo);
}

int do_registra_cliente(struct cliente *cliente)
```

```
{
   MYSQL_BIND param[4];
   int numeroTavolo;

   set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, cliente->nome, strlen(cliente->nome));
   set_binding_param(&param[1], MYSQL_TYPE_VAR_STRING, cliente->cognome, strlen(cliente->cognome));
   set_binding_param(&param[2], MYSQL_TYPE_LONG, &cliente->numeroCommensali, sizeof(cliente->numeroCommensali));
   set_binding_param(&param[3], MYSQL_TYPE_LONG, &numeroTavolo, sizeof(numeroTavolo));

   if (mysql_stmt_bind_param(registra_cliente, param)) {
      print_stmt_error(registra_cliente, "Impossibile effettuare il binding dei parametri per do_registra_cliente");
      numeroTavolo = -1;
      goto out;
   }

   if (mysql_stmt_execute(registra_cliente) != 0) {
      print_stmt_error(registra_cliente, "Impossibile eseguire la procedura 'registra_cliente'");
      numeroTavolo = -1;
      goto out;
   }

   //output
   set_binding_param(&param[0], MYSQL_TYPE_LONG, &numeroTavolo, sizeof(numeroTavolo));

   if(mysql_stmt_bind_result(registra_cliente, param)) {
      print_stmt_error(registra_cliente, "Impossibile recuperare i parametri di output");
      numeroTavolo = -1;
      goto out;
   }

   if(mysql_stmt_fetch(registra_cliente)) {
      print_stmt_error(registra_cliente, "Impossibile bufferizzare l'output");
      numeroTavolo = -1;
      goto out;
   }

   out:
   mysql_stmt_free_result(registra_cliente);
```

```
    while(mysql_stmt_next_result(registra_cliente) != -1) {}
    mysql_stmt_reset(registra_cliente);
    return numeroTavolo;
}

void do_inserisci_pizza_menu(struct pizza *pizza)
{
    MYSQL_BIND param[2];

    set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, pizza->nome, strlen(pizza->nome));
    set_binding_param(&param[1], MYSQL_TYPE_DECIMAL, pizza->prezzo, sizeof(pizza->prezzo));

    if(mysql_stmt_bind_param(inserisci_pizza_menu, param) != 0) {
        print_stmt_error(inserisci_pizza_menu, "Impossibile effettuare il binding dei parametri per do_inserisci_pizza_menu");
        return;
    }

    if (mysql_stmt_execute(inserisci_pizza_menu) != 0) {
        print_stmt_error(inserisci_pizza_menu, "Impossibile eseguire la procedura 'inserisci_pizza_menu'");
        return;
    }
    printf("\nPizza ‘%s’ inserita nel menu\n", pizza->nome);

    mysql_stmt_free_result(inserisci_pizza_menu);
    mysql_stmt_reset(inserisci_pizza_menu);
}

void do_inserisci_ingrediente_menu(struct ingrediente *ingrediente)
{
    MYSQL_BIND param[2];

    set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, ingrediente->nome, strlen(ingrediente->nome));
    set_binding_param(&param[1], MYSQL_TYPE_DECIMAL, ingrediente->prezzo, sizeof(ingrediente->prezzo));

    if(mysql_stmt_bind_param(inserisci_ingrediente_menu, param) != 0) {
        print_stmt_error(inserisci_ingrediente_menu, "Impossibile effettuare il binding dei parametri per do_inserisci_ingrediente_menu");
        return;
```

```
    }

    if (mysql_stmt_execute(inserisci_ingrediente_menu) != 0) {
        print_stmt_error(inserisci_ingrediente_menu, "Impossibile eseguire la procedura
'inserisci_ingrediente_menu'");
        return;
    }
    printf("\nIngrediente '%s' inserito nel menu\n", ingrediente->nome);


    mysql_stmt_free_result(inserisci_ingrediente_menu);
    mysql_stmt_reset(inserisci_ingrediente_menu);
}

bool do_inserisci_condimento_pizza(struct condimento *condimento)
{
    MYSQL_BIND param[2];
    set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, condimento->pizza,
strlen(condimento->pizza));
    set_binding_param(&param[1], MYSQL_TYPE_VAR_STRING, condimento->ingrediente,
strlen(condimento->ingrediente));

    if(mysql_stmt_bind_param(inserisci_condimento_pizza, param) != 0) {
        print_stmt_error(inserisci_condimento_pizza, "Impossibile effettuare il binding dei parametri
per do_inserisci_condimento_pizza");
        return false;
    }

    if (mysql_stmt_execute(inserisci_condimento_pizza) != 0) {
        print_stmt_error(inserisci_condimento_pizza, "Impossibile eseguire la procedura
'inserisci_condimento_pizza'");
        return false;
    }
printf("\nL'ingrediente '%s' e' ora usato come condimento per la pizza '%s'\n", condimento-
>ingrediente, condimento->pizza);


    mysql_stmt_free_result(inserisci_condimento_pizza);
    mysql_stmt_reset(inserisci_condimento_pizza);
    return true;
}

void do_inserisci_bevanda_menu(struct bevanda *bevanda)
{
```

```
    MYSQL_BIND param[2];

    set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, bevanda->nome,
strlen(bevanda->nome));
    set_binding_param(&param[1], MYSQL_TYPE_DECIMAL, bevanda->prezzo, sizeof(bevanda-
>prezzo));

    if(mysql_stmt_bind_param(inserisci_bevanda_menu, param) != 0) {
        print_stmt_error(inserisci_bevanda_menu, "Impossibile effettuare il binding dei parametri per
do_inserisci_bevanda_menu");
        return;
    }

    if (mysql_stmt_execute(inserisci_bevanda_menu) != 0) {
        print_stmt_error(inserisci_bevanda_menu, "Impossibile eseguire la procedura
'inserisci_bevanda_menu'");
        return;
    }
    printf("\nBevanda ‘%s’ inserita nel menu\n", bevanda->nome);

    mysql_stmt_free_result(inserisci_bevanda_menu);
    mysql_stmt_reset(inserisci_bevanda_menu);
}

void do_aggiorna_quantita_ingrediente(struct prodotto_quantita *ingredienteQuantita)
{
    MYSQL_BIND param[2];

    set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, ingredienteQuantita->nome,
strlen(ingredienteQuantita->nome));
    set_binding_param(&param[1], MYSQL_TYPE_LONG, &ingredienteQuantita->quantita,
sizeof(ingredienteQuantita->quantita));

    if(mysql_stmt_bind_param(aggiorna_quantita_ingrediente, param) != 0) {
        print_stmt_error(aggiorna_quantita_ingrediente, "Impossibile effettuare il binding dei parametri
per do_aggiorna_quantita_ingrediente");
        return;
    }

    if (mysql_stmt_execute(aggiorna_quantita_ingrediente) != 0) {
        print_stmt_error(aggiorna_quantita_ingrediente, "Impossibile eseguire la procedura
'aggiorna_quantita_ingrediente'");
        return;
    }
```

```
    printf("\nSono state aggiunte in magazzino %u unita' di %s\n", ingredienteQuantita->quantita,
ingredienteQuantita->nome);

    mysql_stmt_free_result(aggiorna_quantita_ingrediente);
    mysql_stmt_reset(aggiorna_quantita_ingrediente);
}

void do_aggiorna_quantita_bevanda(struct prodotto_quantita *bevandaQuantita)
{
    MYSQL_BIND param[2];

    set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, bevandaQuantita->nome,
strlen(bevandaQuantita->nome));
    set_binding_param(&param[1], MYSQL_TYPE_LONG, &bevandaQuantita->quantita,
sizeof(bevandaQuantita->quantita));

    if(mysql_stmt_bind_param(aggiorna_quantita_bevanda, param) != 0) {
        print_stmt_error(aggiorna_quantita_bevanda, "Impossibile effettuare il binding dei parametri
per do_aggiorna_quantita_bevanda");
        return;
    }

    if (mysql_stmt_execute(aggiorna_quantita_bevanda) != 0) {
        print_stmt_error(aggiorna_quantita_bevanda, "Impossibile eseguire la procedura
'aggiorna_quantita_bevanda'");
        return;
    }
    printf("\nSono state aggiunte in magazzino %u unita' di %s\n", bevandaQuantita->quantita,
bevandaQuantita->nome);


    mysql_stmt_free_result(aggiorna_quantita_bevanda);
    mysql_stmt_reset(aggiorna_quantita_bevanda);
}

struct scontrino *do_stampa_scontrino(struct tavolo *tavolo)
{
    MYSQL_BIND param[3];
    MYSQL_TIME dataOra;
    char prezzo[PREZZO_SCONTRINO_LEN];
    struct scontrino *scontrino = NULL;

    init_mysql_timestamp(&dataOra);
```

```
    set_binding_param(&param[0], MYSQL_TYPE_LONG, &tavolo->numeroTavolo, sizeof(tavolo->numeroTavolo));
    set_binding_param(&param[1], MYSQL_TYPE_DECIMAL, prezzo, strlen(prezzo));
    set_binding_param(&param[2], MYSQL_TYPE_DATETIME, &dataOra, sizeof(dataOra));

    if (mysql_stmt_bind_param(stampa_scontrino, param)) {
        print_stmt_error(stampa_scontrino, "Impossibile effettuare il binding dei parametri per do_stampa_scontrino");
        goto out;
    }

    if (mysql_stmt_execute(stampa_scontrino) != 0) {
        print_stmt_error(stampa_scontrino, "Impossibile eseguire la procedura 'stampa_scontrino'");
        goto out;
    }

    scontrino = malloc(sizeof *scontrino);
    if(scontrino == NULL)
        goto out;
    memset(scontrino, 0, sizeof(*scontrino));

    //output
    set_binding_param(&param[0], MYSQL_TYPE_DECIMAL, prezzo, PREZZO_SCONTRINO_LEN);
    set_binding_param(&param[1], MYSQL_TYPE_DATETIME, &dataOra, sizeof(dataOra));

    if(mysql_stmt_bind_result(stampa_scontrino, param)) {
        print_stmt_error(stampa_scontrino, "Impossibile recuperare i parametri di output");
        free(scontrino);
        scontrino = NULL;
        goto out;
    }

    if(mysql_stmt_fetch(stampa_scontrino)) {
        print_stmt_error(stampa_scontrino, "Impossibile bufferizzare l'output");
        goto out;
    }
    strcpy(scontrino->prezzo, prezzo);
    mysql_timestamp_to_string(&dataOra, scontrino->id);

    out:
    mysql_stmt_free_result(stampa_scontrino);
    while(mysql_stmt_next_result(stampa_scontrino) != -1) {}
    mysql_stmt_reset(stampa_scontrino);
```

```
    return scontrino;
}

void do_registra_pagamento_scontrino(struct scontrino *scontrino)
{
    MYSQL_BIND param[1];
    MYSQL_TIME dataOra;

    datetime_to_mysql_time(scontrino->id, &dataOra);
    set_binding_param(&param[0], MYSQL_TYPE_DATETIME, &dataOra, sizeof(dataOra));

    if(mysql_stmt_bind_param(registra_pagamento_scontrino, param) != 0) {
        print_stmt_error(registra_pagamento_scontrino, "Impossibile effettuare il binding dei parametri
per do_registra_pagamento_scontrino");
        return;
    }

    if (mysql_stmt_execute(registra_pagamento_scontrino) != 0) {
        print_stmt_error(registra_pagamento_scontrino, "Impossibile eseguire la procedura
'registra_pagamento_scontrino'");
        return;
    }
    puts("\nIl pagamento dello scontrino e' andato a buon fine");


    mysql_stmt_free_result(registra_pagamento_scontrino);
    mysql_stmt_reset(registra_pagamento_scontrino);
}

struct entrata *do_visualizza_entrata_giornaliera(char giorno[DATE_LEN])
{
    MYSQL_BIND param[2];
    MYSQL_TIME data;
    char entrata[ENTRATE_LEN];
    struct entrata *entrataGiornaliera  = NULL;

    date_to_mysql_time(giorno, &data);

    set_binding_param(&param[0], MYSQL_TYPE_DATE, &data, sizeof(data));
    set_binding_param(&param[1], MYSQL_TYPE_DECIMAL, entrata, strlen(entrata));

    if (mysql_stmt_bind_param(visualizza_entrata_giornaliera, param)) {
        print_stmt_error(visualizza_entrata_giornaliera, "Impossibile effettuare il binding dei parametri
per do_visualizza_entrata_giornaliera");
```

```
        goto out;
    }

    if (mysql_stmt_execute(visualizza_entrata_giornaliera) != 0) {
        print_stmt_error(visualizza_entrata_giornaliera, "Impossibile eseguire la procedura
'visualizza_entrata_giornaliera'");
        goto out;
    }

    entrataGiornaliera = malloc(sizeof *entrataGiornaliera);
    if(entrataGiornaliera == NULL)
        goto out;
    memset(entrataGiornaliera, 0, sizeof(*entrataGiornaliera));

    //output
    set_binding_param(&param[0], MYSQL_TYPE_DECIMAL, entrata, ENTRATE_LEN);

    if(mysql_stmt_bind_result(visualizza_entrata_giornaliera, param)) {
        print_stmt_error(visualizza_entrata_giornaliera, "Impossibile recuperare i parametri di output");
        free(entrataGiornaliera);
        entrataGiornaliera = NULL;
        goto out;
    }

    if(mysql_stmt_fetch(visualizza_entrata_giornaliera)) {
        print_stmt_error(visualizza_entrata_giornaliera, "Impossibile bufferizzare l'output");
        goto out;
    }
    strcpy(entrataGiornaliera->tot, entrata);

    out:
    mysql_stmt_free_result(visualizza_entrata_giornaliera);
    while(mysql_stmt_next_result(visualizza_entrata_giornaliera) != -1) {}
    mysql_stmt_reset(visualizza_entrata_giornaliera);
    return entrataGiornaliera;
}

struct entrata *do_visualizza_entrata_mensile(struct anno_mese *annoMese)
{
    MYSQL_BIND param[3];
    char entrata[ENTRATE_LEN];
    struct entrata *entrataMensile  = NULL;

    set_binding_param(&param[0], MYSQL_TYPE_SHORT, &annoMese->anno, sizeof(annoMese-
```

```
>anno));
   set_binding_param(&param[1], MYSQL_TYPE_LONG, &annoMese->mese, sizeof(annoMese-
>mese));
   set_binding_param(&param[2], MYSQL_TYPE_DECIMAL, entrata, strlen(entrata));

   if (mysql_stmt_bind_param(visualizza_entrata_mensile, param)) {
      print_stmt_error(visualizza_entrata_mensile, "Impossibile effettuare il binding dei parametri per
do_visualizza_entrata_mensile");
      goto out;
   }

   if (mysql_stmt_execute(visualizza_entrata_mensile) != 0) {
      print_stmt_error(visualizza_entrata_mensile, "Impossibile eseguire la procedura
'visualizza_entrata_mensile'");
      goto out;
   }

   entrataMensile = malloc(sizeof *entrataMensile);
   if(entrataMensile == NULL)
      goto out;
   memset(entrataMensile, 0, sizeof(*entrataMensile));

   //output
   set_binding_param(&param[0], MYSQL_TYPE_DECIMAL, entrata, ENTRATE_LEN);

   if(mysql_stmt_bind_result(visualizza_entrata_mensile, param)) {
      print_stmt_error(visualizza_entrata_mensile, "Impossibile recuperare i parametri di output");
      free(entrataMensile);
      entrataMensile = NULL;
      goto out;
   }

   if(mysql_stmt_fetch(visualizza_entrata_mensile)) {
      print_stmt_error(visualizza_entrata_mensile, "Impossibile bufferizzare l'output");
      goto out;
   }
   strcpy(entrataMensile->tot, entrata);

   out:
   mysql_stmt_free_result(visualizza_entrata_mensile);
   while(mysql_stmt_next_result(visualizza_entrata_mensile) != -1) {}
   mysql_stmt_reset(visualizza_entrata_mensile);
   return entrataMensile;
}
```

```
void do_inserisci_tavolo(struct nuovo_tavolo *nuovoTavolo)
{
   MYSQL_BIND param[2];

   set_binding_param(&param[0], MYSQL_TYPE_LONG, &nuovoTavolo->numeroTavolo,
sizeof(nuovoTavolo->numeroTavolo));
   set_binding_param(&param[1], MYSQL_TYPE_LONG, &nuovoTavolo->numeroPosti, sizeof
(nuovoTavolo->numeroPosti));

   if(mysql_stmt_bind_param(inserisci_tavolo, param) != 0) {
      print_stmt_error(inserisci_tavolo, "Impossibile effettuare il binding dei parametri per
do_inserisci_tavolo");
      return;
   }

   //esecuzione procedura
   if(mysql_stmt_execute(inserisci_tavolo) != 0) {
      print_stmt_error(inserisci_tavolo, "Impossibile eseguire la procedura 'inserisci_tavolo'");
      return;
   }
   printf("\nIl tavolo %u e' stato inserito correttamente nel sistema\n", nuovoTavolo-
>numeroTavolo);

   mysql_stmt_free_result(inserisci_tavolo);
   mysql_stmt_reset(inserisci_tavolo);
}

void do_inserisci_cameriere(struct nuovo_cameriere *nuovoCameriere)
{
   MYSQL_BIND param[2];

   set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, nuovoCameriere->nome,
strlen(nuovoCameriere->nome));
   set_binding_param(&param[1], MYSQL_TYPE_VAR_STRING, nuovoCameriere->cognome,
strlen(nuovoCameriere->cognome));

   if(mysql_stmt_bind_param(inserisci_cameriere, param) != 0) {
      print_stmt_error(inserisci_cameriere, "Impossibile effettuare il binding dei parametri per
do_inserisci_cameriere");
      return;
   }

   //esecuzione procedura
```

```
    if(mysql_stmt_execute(inserisci_cameriere) != 0) {
        print_stmt_error(inserisci_cameriere, "Impossibile eseguire la procedura 'inserisci_cameriere'");
        return;
    }
    printf("\nIl cameriere %s %s e' stato inserito correttamente nel sistema\n", nuovoCameriere->nome, nuovoCameriere->cognome);

    mysql_stmt_free_result(inserisci_cameriere);
    mysql_stmt_reset(inserisci_cameriere);
}

void do_cancella_turno_cameriere(struct turno_cameriere *turnoCameriere)
{
    MYSQL_BIND param[4];
    MYSQL_TIME data;

    date_to_mysql_time(turnoCameriere->data, &data);

    set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, turnoCameriere->nomeC, strlen(turnoCameriere->nomeC));
    set_binding_param(&param[1], MYSQL_TYPE_VAR_STRING, turnoCameriere->cognomeC, strlen(turnoCameriere->cognomeC));
    set_binding_param(&param[2], MYSQL_TYPE_VAR_STRING, turnoCameriere->turno, strlen(turnoCameriere->turno));
    set_binding_param(&param[3], MYSQL_TYPE_DATE, &data, sizeof(data));

    if(mysql_stmt_bind_param(cancella_turno_cameriere, param) != 0) {
        print_stmt_error(cancella_turno_cameriere, "Impossibile effettuare il binding dei parametri per do_cancella_turno_cameriere");
        return;
    }

    //esecuzione procedura
    if(mysql_stmt_execute(cancella_turno_cameriere) != 0) {
        print_stmt_error(cancella_turno_cameriere, "Impossibile eseguire la procedura 'cancella_turno_cameriere'");
        return;
    }
    puts("\nModifica eseguita correttamente");

    mysql_stmt_free_result(cancella_turno_cameriere);
    mysql_stmt_reset(cancella_turno_cameriere);
}
```

```
//------------------ CAMERIERE ------------------

struct lista_tavoli *do_visualizza_tavoli_occupati(char *username)
{
   int status;
   size_t row = 0;
   MYSQL_BIND param[1];
   unsigned int tavolo;
   struct lista_tavoli *tavoliOccupati = NULL;

   set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, username, strlen(username));

   if (mysql_stmt_bind_param(visualizza_tavoli_occupati, param)) {
      print_stmt_error(visualizza_tavoli_occupati, "Impossibile effettuare il binding dei parametri per
do_visualizza_tavoli_occupati");
      goto out;
   }

   if (mysql_stmt_execute(visualizza_tavoli_occupati) != 0) {
      print_stmt_error(visualizza_tavoli_occupati, "Impossibile eseguire la procedura
'visualizza_tavoli_occupati'");
      goto out;
   }

   mysql_stmt_store_result(visualizza_tavoli_occupati);

   tavoliOccupati = malloc(sizeof (*tavoliOccupati) + sizeof(struct tavolo) *
mysql_stmt_num_rows(visualizza_tavoli_occupati));
   if(tavoliOccupati == NULL)
      goto out;
   memset(tavoliOccupati, 0, sizeof(*tavoliOccupati) + sizeof(struct tavolo) *
mysql_stmt_num_rows(visualizza_tavoli_occupati));
   tavoliOccupati->num_entries = mysql_stmt_num_rows(visualizza_tavoli_occupati);


   set_binding_param(&param[0], MYSQL_TYPE_LONG, &tavolo, sizeof(tavolo));

   if(mysql_stmt_bind_result(visualizza_tavoli_occupati, param)) {
      print_stmt_error(visualizza_tavoli_occupati, "Impossibile effettuare il bind dei parametri per
visualizzare i tavoli occupati\n");
      free(tavoliOccupati);
      tavoliOccupati = NULL;
      goto out;
   }
```

```
    while (true) {
        status = mysql_stmt_fetch(visualizza_tavoli_occupati);

        if (status == 1 || status == MYSQL_NO_DATA)
            break;

        tavoliOccupati->tavolo[row].numeroTavolo = tavolo;

        row++;
    }
    out:
    mysql_stmt_free_result(visualizza_tavoli_occupati);
    while(mysql_stmt_next_result(visualizza_tavoli_occupati) != -1) {}
    mysql_stmt_reset(visualizza_tavoli_occupati);
    return tavoliOccupati;
}

struct lista_tavoli *do_visualizza_tavoli_comande_servite(char *username)
{
    int status;
    size_t row = 0;
    MYSQL_BIND param[1];
    unsigned int tavolo;
    unsigned int num_tavoli;
    struct lista_tavoli *tavoliComandeServite = NULL;

    set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, username, strlen(username));

    if (mysql_stmt_bind_param(visualizza_tavoli_comande_servite, param)) {
        print_stmt_error(visualizza_tavoli_comande_servite, "Impossibile effettuare il binding dei
parametri per do_visualizza_tavoli_comande_servite");
        goto out;
    }

    if (mysql_stmt_execute(visualizza_tavoli_comande_servite) != 0) {
        print_stmt_error(visualizza_tavoli_comande_servite, "Impossibile eseguire la procedura
'visualizza_tavoli_comande_servite'");
        goto out;
    }

    mysql_stmt_store_result(visualizza_tavoli_comande_servite);
    num_tavoli = mysql_stmt_num_rows(visualizza_tavoli_comande_servite);
```

```
    tavoliComandeServite = malloc(sizeof (*tavoliComandeServite) + sizeof(struct tavolo) *
num_tavoli);
    if(tavoliComandeServite == NULL)
        goto out;
    memset(tavoliComandeServite, 0, sizeof(*tavoliComandeServite) + sizeof(struct tavolo) *
num_tavoli);
    tavoliComandeServite->num_entries = num_tavoli;

    set_binding_param(&param[0], MYSQL_TYPE_LONG, &tavolo, sizeof(tavolo));

    if(mysql_stmt_bind_result(visualizza_tavoli_comande_servite, param)) {
        print_stmt_error(visualizza_tavoli_comande_servite, "Impossibile effettuare il bind dei
parametri per visualizzare i tavoli in cui le comande sono state completamente servite\n");
        free(tavoliComandeServite);
        tavoliComandeServite = NULL;
        goto out;
    }

    while (true) {
        status = mysql_stmt_fetch(visualizza_tavoli_comande_servite);

        if (status == 1 || status == MYSQL_NO_DATA)
            break;

        tavoliComandeServite->tavolo[row].numeroTavolo = tavolo;

        row++;
    }
    out:
    mysql_stmt_free_result(visualizza_tavoli_comande_servite);
    while(mysql_stmt_next_result(visualizza_tavoli_comande_servite) != -1) {}
    mysql_stmt_reset(visualizza_tavoli_comande_servite);
    return tavoliComandeServite;
}


static struct da_servire *extract_comande_da_servire_info(void)
{
    struct da_servire *daServire = NULL;
    int stato;
    size_t row = 0;
    MYSQL_BIND param[2];
    unsigned int comanda;
    unsigned int tavolo;
```

```
    unsigned int num_comande;

    set_binding_param(&param[0], MYSQL_TYPE_LONG, &comanda, sizeof(comanda));
    set_binding_param(&param[1], MYSQL_TYPE_LONG, &tavolo, sizeof(tavolo));

    if(mysql_stmt_bind_result(visualizza_cosa_e_dove_servire, param)) {
        print_stmt_error(visualizza_cosa_e_dove_servire, "Impossibile effettuare il bind dei parametri\
n");
        free(daServire);
        daServire = NULL;
        goto out;
    }

    if (mysql_stmt_store_result(visualizza_cosa_e_dove_servire)) {
        print_stmt_error(visualizza_cosa_e_dove_servire, "Impossibile bufferizzare l'intero result set");
        goto out;
    }
    num_comande = mysql_stmt_num_rows(visualizza_cosa_e_dove_servire);

    daServire = malloc(sizeof(*daServire) + sizeof(struct comande_da_servire) * num_comande);
    if(daServire == NULL)
        goto out;

    memset(daServire, 0, sizeof(*daServire) + sizeof(struct comande_da_servire) * num_comande);

    daServire->num_comande = num_comande;

    while (true) {
        stato = mysql_stmt_fetch(visualizza_cosa_e_dove_servire);

        if (stato == 1 || stato == MYSQL_NO_DATA) {
            break;
        }

        daServire->comandeDaServire[row].comanda = comanda;
        daServire->comandeDaServire[row].tavolo = tavolo;

        row++;
    }
    out:
    return daServire;
}

static void *extract_prodotti_da_servire_info(struct da_servire *daServire, size_t i)
```

```
{
    int stato;
    size_t row = 0;
    MYSQL_BIND param[2];
    unsigned int id;
    char nome[NOME_PRODOTTO_LEN];
    unsigned int num_prodotti;

    assert(i < daServire->num_comande);

    set_binding_param(&param[0], MYSQL_TYPE_LONG, &id, sizeof(id));
    set_binding_param(&param[1], MYSQL_TYPE_VAR_STRING, nome,
NOME_PRODOTTO_LEN);

    if(mysql_stmt_bind_result(visualizza_cosa_e_dove_servire, param)) {
        finish_with_stmt_error(conn, visualizza_cosa_e_dove_servire, "[FATAL] Impossibile
bufferizzare i parametri\n", false);
    }

    mysql_stmt_store_result(visualizza_cosa_e_dove_servire);
    num_prodotti = mysql_stmt_num_rows(visualizza_cosa_e_dove_servire);

    daServire->comandeDaServire[i].prodottiDaServire = malloc(sizeof(*daServire-
>comandeDaServire[i].prodottiDaServire) * num_prodotti);
    daServire->comandeDaServire[i].num_prodotti = num_prodotti;

    while (true) {
        stato = mysql_stmt_fetch(visualizza_cosa_e_dove_servire);

        if (stato == 1 || stato == MYSQL_NO_DATA) {
            break;
        }

        daServire->comandeDaServire[i].prodottiDaServire[row].id = id;
        strcpy(daServire->comandeDaServire[i].prodottiDaServire[row].nome, nome);

        row++;
    }
}

struct da_servire *do_visualizza_cosa_e_dove_servire(char *username)
{
    int stato;
    unsigned incr = 0;
```

```
    MYSQL_BIND param[1];
    struct da_servire *daServire = NULL;

    set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, username, strlen(username));

    if(mysql_stmt_bind_param(visualizza_cosa_e_dove_servire, param) != 0) {
        finish_with_stmt_error(conn, visualizza_cosa_e_dove_servire, "[FATAL] Impossibile
bufferizzare i parametri\n", false);
        goto out;
    }

    if(mysql_stmt_execute(visualizza_cosa_e_dove_servire) != 0) {
        print_stmt_error(visualizza_cosa_e_dove_servire, "Impossibile eseguire la procedura per
ottenere le cose da servire\n");
        goto out;
    }

    do {
        // vedo se la procedura contiene dei valori out o inout tramite il bit
SERVER_PS_OUT_PARAMS,
        // in caso li abbia il result set che precede il valore dello stato finale (questo lo trovo facendo
conn->server_status)
        //conterra' il valore dei parametri di output

        if (conn->server_status & SERVER_PS_OUT_PARAMS) {
            goto next;
        }
        if (incr == 0) {
            daServire = extract_comande_da_servire_info();
            if (daServire == NULL) {
                goto out;
            }
        } else if (incr - 1 < daServire->num_comande)
            extract_prodotti_da_servire_info(daServire, incr - 1);

        next:
        mysql_stmt_free_result(visualizza_cosa_e_dove_servire);
        stato = mysql_stmt_next_result(visualizza_cosa_e_dove_servire);
        if (stato > 0)
            finish_with_stmt_error(conn, visualizza_cosa_e_dove_servire, "Condizione inaspettata",
false);
        incr++;
    } while (stato == 0);
```

```
    out:
    mysql_stmt_free_result(visualizza_cosa_e_dove_servire);
    mysql_stmt_reset(visualizza_cosa_e_dove_servire);
    return daServire;
}

void free_da_servire(struct da_servire *daServire)
{
    for (size_t i = 0; i < daServire->num_comande; i++) {
        free(daServire->comandeDaServire[i].prodottiDaServire);
    }
    free(daServire);
}


int do_registra_comanda(struct tavolo *tavolo, char *username)
{
    MYSQL_BIND param[3];
    int numeroComanda;

    set_binding_param(&param[0], MYSQL_TYPE_LONG, &tavolo->numeroTavolo, sizeof(tavolo-
>numeroTavolo));
    set_binding_param(&param[1], MYSQL_TYPE_LONG, &numeroComanda,
sizeof(numeroComanda));
    set_binding_param(&param[2], MYSQL_TYPE_VAR_STRING, username, strlen(username));

    if (mysql_stmt_bind_param(registra_comanda, param)) {
        print_stmt_error(registra_comanda, "Impossibile effettuare il binding dei parametri per
do_registra_comanda");
        numeroComanda = -1;
        goto out;
    }

    if (mysql_stmt_execute(registra_comanda) != 0) {
        print_stmt_error(registra_comanda, "Impossibile eseguire la procedura 'registra_comanda'");
        numeroComanda = -1;
        goto out;
    }

    //output
    set_binding_param(&param[0], MYSQL_TYPE_LONG, &numeroComanda,
sizeof(numeroComanda));

    if(mysql_stmt_bind_result(registra_comanda, param)) {
```

```
        print_stmt_error(registra_comanda, "Impossibile recuperare i parametri di output");
        numeroComanda = -1;
        goto out;
    }

    if(mysql_stmt_fetch(registra_comanda)) {
        print_stmt_error(registra_comanda, "Impossibile bufferizzare l'output");
        numeroComanda = -1;
        goto out;
    }

    out:
    mysql_stmt_free_result(registra_comanda);
    while(mysql_stmt_next_result(registra_comanda) != -1) {}
    mysql_stmt_reset(registra_comanda);
    return numeroComanda;
}


bool do_ordina_bevanda(struct bevanda_effettiva *bevandaEffettiva, char *username)
{
    MYSQL_BIND param[3];

    set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, bevandaEffettiva-
>nomeBevanda, strlen(bevandaEffettiva->nomeBevanda));
    set_binding_param(&param[1], MYSQL_TYPE_LONG, &bevandaEffettiva->comanda,
sizeof(bevandaEffettiva->comanda));
    set_binding_param(&param[2], MYSQL_TYPE_VAR_STRING, username, strlen(username));

    if(mysql_stmt_bind_param(ordina_bevanda, param) != 0) {
        print_stmt_error(ordina_bevanda, "Impossibile effettuare il binding dei parametri per
do_ordina_bevanda");
        return false;
    }

    if (mysql_stmt_execute(ordina_bevanda) != 0) {
        print_stmt_error(ordina_bevanda, "Impossibile eseguire la procedura 'ordina_bevanda'");
        return false;
    }
    printf("\nOrdinazione della bevanda '%s' effettuata correttamente\n", bevandaEffettiva-
>nomeBevanda);


    mysql_stmt_free_result(ordina_bevanda);
```

```
    mysql_stmt_reset(ordina_bevanda);
    return true;
}

bool do_segna_bevanda_servita(struct bevanda_stato *bevandaServita, char *username)
{
    MYSQL_BIND param[3];

    set_binding_param(&param[0], MYSQL_TYPE_LONG, &bevandaServita->idBevanda,
sizeof(bevandaServita->idBevanda));
    set_binding_param(&param[1], MYSQL_TYPE_LONG, &bevandaServita->comanda,
sizeof(bevandaServita->comanda));
    set_binding_param(&param[2], MYSQL_TYPE_VAR_STRING, username, strlen(username));

    if(mysql_stmt_bind_param(segna_bevanda_servita, param) != 0) {
        print_stmt_error(segna_bevanda_servita, "Impossibile effettuare il binding dei parametri per
do_segna_bevanda_servita");
        return false;
    }

    if (mysql_stmt_execute(segna_bevanda_servita) != 0) {
        print_stmt_error(segna_bevanda_servita, "Impossibile eseguire la procedura
'segna_bevanda_servita'");
        return false;
    }
    puts("\nAggiornamento dello stato della bevanda andato a buon fine");


    mysql_stmt_free_result(segna_bevanda_servita);
    mysql_stmt_reset(segna_bevanda_servita);
    return true;
}

int do_ordina_pizza(struct pizza_effettiva *pizzaEffettiva, char *username)
{
    MYSQL_BIND param[4];
    int idPizza;

    set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, pizzaEffettiva->nomePizza,
strlen(pizzaEffettiva->nomePizza));
    set_binding_param(&param[1], MYSQL_TYPE_LONG, &pizzaEffettiva->comanda,
sizeof(pizzaEffettiva->comanda));
    set_binding_param(&param[2], MYSQL_TYPE_VAR_STRING, username, strlen(username));
    set_binding_param(&param[3], MYSQL_TYPE_LONG, &idPizza, sizeof(idPizza));
```

```
    if (mysql_stmt_bind_param(ordina_pizza, param)) {
        print_stmt_error(ordina_pizza, "Impossibile effettuare il binding dei parametri per
do_ordina_pizza");
        idPizza = -1;
        goto out;
    }

    if (mysql_stmt_execute(ordina_pizza) != 0) {
        print_stmt_error(ordina_pizza, "Impossibile eseguire la procedura 'ordina_pizza'");
        idPizza = -1;
        goto out;
    }

    //output
    set_binding_param(&param[0], MYSQL_TYPE_LONG, &idPizza, sizeof(idPizza));

    if(mysql_stmt_bind_result(ordina_pizza, param)) {
        print_stmt_error(ordina_pizza, "Impossibile recuperare i parametri di output");
        idPizza = -1;
        goto out;
    }

    if(mysql_stmt_fetch(ordina_pizza)) {
        print_stmt_error(ordina_pizza, "Impossibile bufferizzare l'output");
        idPizza = -1;
        goto out;
    }

    out:
    mysql_stmt_free_result(ordina_pizza);
    while(mysql_stmt_next_result(ordina_pizza) != -1) {}
    mysql_stmt_reset(ordina_pizza);
    return idPizza;
}

void do_inserisci_aggiunta_pizza(struct aggiunta_pizza *aggiunta)
{
    MYSQL_BIND param[3];

    set_binding_param(&param[0], MYSQL_TYPE_LONG, &aggiunta->idPizza, sizeof(aggiunta-
>idPizza));
    set_binding_param(&param[1], MYSQL_TYPE_LONG, &aggiunta->comanda, sizeof(aggiunta-
>comanda));
```

```
    set_binding_param(&param[2], MYSQL_TYPE_VAR_STRING, aggiunta->ingrediente,
strlen(aggiunta->ingrediente));

    if(mysql_stmt_bind_param(inserisci_aggiunta_pizza, param) != 0) {
        print_stmt_error(inserisci_aggiunta_pizza, "Impossibile effettuare il binding dei parametri per
do_inserisci_aggiunta_pizza");
        return;
    }

    if (mysql_stmt_execute(inserisci_aggiunta_pizza) != 0) {
        print_stmt_error(inserisci_aggiunta_pizza, "Impossibile eseguire la procedura
'inserisci_aggiunta_pizza'");
        return;
    }
    printf("\nL'ingrediente '%s' e' stato aggiunto alla pizza %u\n", aggiunta->ingrediente, aggiunta-
>idPizza);

    mysql_stmt_free_result(inserisci_aggiunta_pizza);
    mysql_stmt_reset(inserisci_aggiunta_pizza);
}

void do_segna_pizza_ordinata(struct pizza_stato *pizzaOrdinata)
{
    MYSQL_BIND param[2];

    set_binding_param(&param[0], MYSQL_TYPE_LONG, &pizzaOrdinata->idPizza,
sizeof(pizzaOrdinata->idPizza));
    set_binding_param(&param[1], MYSQL_TYPE_LONG, &pizzaOrdinata->comanda,
sizeof(pizzaOrdinata->comanda));

    if(mysql_stmt_bind_param(segna_pizza_ordinata, param) != 0) {
        print_stmt_error(segna_pizza_ordinata, "Impossibile effettuare il binding dei parametri per
do_segna_pizza_ordinata");
        return;
    }

    if (mysql_stmt_execute(segna_pizza_ordinata) != 0) {
        print_stmt_error(segna_pizza_ordinata, "Impossibile eseguire la procedura
'segna_pizza_ordinata'");
        return;
    }
    printf("\nL'ordinazione della pizza %u e' andata a buon fine\n", pizzaOrdinata->idPizza);

    mysql_stmt_free_result(segna_pizza_ordinata);
```

```
    mysql_stmt_reset(segna_pizza_ordinata);
}

bool do_segna_pizza_servita(struct pizza_stato *pizzaServita, char *username)
{
    MYSQL_BIND param[3];

    set_binding_param(&param[0], MYSQL_TYPE_LONG, &pizzaServita->idPizza,
sizeof(pizzaServita->idPizza));
    set_binding_param(&param[1], MYSQL_TYPE_LONG, &pizzaServita->comanda,
sizeof(pizzaServita->comanda));
    set_binding_param(&param[2], MYSQL_TYPE_VAR_STRING, username, strlen(username));

    if(mysql_stmt_bind_param(segna_pizza_servita, param) != 0) {
        print_stmt_error(segna_pizza_servita, "Impossibile effettuare il binding dei parametri per
do_segna_pizza_servita");
        return false;
    }

    if (mysql_stmt_execute(segna_pizza_servita) != 0) {
        print_stmt_error(segna_pizza_servita, "Impossibile eseguire la procedura
'segna_pizza_servita'");
        return false;
    }
    puts("\nAggiornamneto dello stato della pizza andato a buon fine");


    mysql_stmt_free_result(segna_pizza_servita);
    mysql_stmt_reset(segna_pizza_servita);
    return true;
}



//---------------PIZZAIOLO---------------------------------

static struct pizze_da_preparare *extract_pizze_da_preparare_info(void)
{
    struct pizze_da_preparare *pizzeDaPreparare = NULL;
    int stato;
    size_t row = 0;
    MYSQL_BIND param[3];
    unsigned int comanda;
    unsigned int idPizza;
```

```
    char nomePizza[NOME_PRODOTTO_LEN];
    unsigned int num_pizze;

    set_binding_param(&param[0], MYSQL_TYPE_LONG, &comanda, sizeof(comanda));
    set_binding_param(&param[1], MYSQL_TYPE_LONG, &idPizza, sizeof(idPizza));
    set_binding_param(&param[2], MYSQL_TYPE_VAR_STRING, nomePizza,
NOME_PRODOTTO_LEN);

    if(mysql_stmt_bind_result(visualizza_pizze_da_preparare, param)) {
        print_stmt_error(visualizza_pizze_da_preparare, "Impossibile effettuare il bind dei parametri\
n");
        free(pizzeDaPreparare);
        pizzeDaPreparare = NULL;
        goto out;
    }

    if (mysql_stmt_store_result(visualizza_pizze_da_preparare)) {
        print_stmt_error(visualizza_pizze_da_preparare, "Impossibile bufferizzare l'intero result set");
        goto out;
    }
    num_pizze = mysql_stmt_num_rows(visualizza_pizze_da_preparare);

    pizzeDaPreparare = malloc(sizeof(*pizzeDaPreparare) + sizeof(struct pizza_info)  * num_pizze);
    if(pizzeDaPreparare == NULL)
        goto out;

    memset(pizzeDaPreparare, 0, sizeof(*pizzeDaPreparare) + sizeof(struct pizza_info) *
num_pizze);

    pizzeDaPreparare->num_pizze = num_pizze;

    while (true) {
        stato = mysql_stmt_fetch(visualizza_pizze_da_preparare);

        if (stato == 1 || stato == MYSQL_NO_DATA) {
            break;
        }

        pizzeDaPreparare->pizzaInfo[row].comanda = comanda;
        pizzeDaPreparare->pizzaInfo[row].idPizza = idPizza;
        strcpy(pizzeDaPreparare->pizzaInfo[row].nomePizza, nomePizza);

        row++;
    }
```

```
    out:
    return pizzeDaPreparare;
}

static void extract_aggiunte_info(struct pizze_da_preparare *pizzeDaPreparare, size_t i)
{
    MYSQL_BIND param[3];
    size_t row = 0;
    int stato;
    char ingrediente[NOME_PRODOTTO_LEN];

    assert(i < pizzeDaPreparare->num_pizze);

    set_binding_param(&param[0], MYSQL_TYPE_VAR_STRING, ingrediente,
NOME_PRODOTTO_LEN);

    if(mysql_stmt_bind_result(visualizza_pizze_da_preparare, param)) {
        finish_with_stmt_error(conn, visualizza_pizze_da_preparare, "[FATAL] Impossibile
bufferizzare i parametri\n", false);
    }
    mysql_stmt_store_result(visualizza_pizze_da_preparare);

    pizzeDaPreparare->pizzaInfo[i].aggiunte = malloc(sizeof(*pizzeDaPreparare-
>pizzaInfo[i].aggiunte) *
                                        mysql_stmt_num_rows(visualizza_pizze_da_preparare));
    pizzeDaPreparare->pizzaInfo[i].num_aggiunte =
mysql_stmt_num_rows(visualizza_pizze_da_preparare);

    while (true) {
        stato = mysql_stmt_fetch(visualizza_pizze_da_preparare);

        if (stato == 1 || stato == MYSQL_NO_DATA)
            break;

        strcpy(pizzeDaPreparare->pizzaInfo[i].aggiunte[row].ingrediente, ingrediente);

        row++;
    }
}

struct pizze_da_preparare *do_visualizza_pizze_da_preparare(void)
{
    int stato;
    unsigned incr = 0;
```

```
    struct pizze_da_preparare *pizzeDaPreparare = NULL;

    if(mysql_stmt_execute(visualizza_pizze_da_preparare) != 0) {
        print_stmt_error(visualizza_pizze_da_preparare, "Impossibile eseguire la procedura per ottenere
le pizze da preparare\n");
        goto out;
    }
    do {
        if (conn->server_status & SERVER_PS_OUT_PARAMS) {
            goto next;
        }
        if (incr == 0) {
            pizzeDaPreparare = extract_pizze_da_preparare_info();
            if (pizzeDaPreparare == NULL) {
                goto out;
            }
        } else if (incr - 1 < pizzeDaPreparare->num_pizze) {
            extract_aggiunte_info(pizzeDaPreparare, incr - 1);
        }
        next:
        mysql_stmt_free_result(visualizza_pizze_da_preparare);
        stato = mysql_stmt_next_result(visualizza_pizze_da_preparare);
        if (stato > 0)
            finish_with_stmt_error(conn, visualizza_pizze_da_preparare, "Condizione inaspettata",
false);
        incr++;
    } while (stato == 0);

    out:
    mysql_stmt_free_result(visualizza_pizze_da_preparare);
    mysql_stmt_reset(visualizza_pizze_da_preparare);
    return pizzeDaPreparare;
}

void free_pizze_da_preparare(struct pizze_da_preparare *pizzeDaPreparare)
{
    for (size_t i = 0; i < pizzeDaPreparare->num_pizze; i++) {
        free(pizzeDaPreparare->pizzaInfo[i].aggiunte);
    }
    free(pizzeDaPreparare);
}
```

```
bool do_segna_pizza_pronta(struct pizza_stato *pizzaPronta)
{
   MYSQL_BIND param[2];

   set_binding_param(&param[0], MYSQL_TYPE_LONG, &pizzaPronta->idPizza,
sizeof(pizzaPronta->idPizza));
   set_binding_param(&param[1], MYSQL_TYPE_LONG, &pizzaPronta->comanda,
sizeof(pizzaPronta->comanda));

   if(mysql_stmt_bind_param(segna_pizza_pronta, param) != 0) {
      print_stmt_error(segna_pizza_pronta, "Impossibile effettuare il binding dei parametri per
do_segna_pizza_pronta");
      return false;
   }

   if (mysql_stmt_execute(segna_pizza_pronta) != 0) {
      print_stmt_error(segna_pizza_pronta, "Impossibile eseguire la procedura
'segna_pizza_pronta'");
      return false;
   }
   puts("\nAggiornamento dello stato della pizza andato a buon fine");


   mysql_stmt_free_result(segna_pizza_pronta);
   mysql_stmt_reset(segna_pizza_pronta);
   return true;
}



//----------------BARISTA--------------------------------

struct lista_bevande_da_preparare *do_visualizza_bevande_da_preparare(void)
{
   int status;
   size_t row = 0;
   MYSQL_BIND param[3];
   unsigned int comanda;
   unsigned int idBevanda;
   char nomeBevanda[NOME_PRODOTTO_LEN];
   struct lista_bevande_da_preparare *listaBevandeDaPreparare = NULL;

   if (mysql_stmt_execute(visualizza_bevande_da_preparare) != 0) {
      finish_with_stmt_error(conn, visualizza_bevande_da_preparare, "Impossibile eseguire la
```

```
procedura 'visualizza_bevande_da_preparare'", false);
        goto out;
    }

    if(mysql_stmt_store_result(visualizza_bevande_da_preparare)) {
        print_stmt_error(visualizza_bevande_da_preparare, "Impossibile memorizzare la lista di
bevande");
        goto out;
    }

    listaBevandeDaPreparare = malloc(sizeof (*listaBevandeDaPreparare) + sizeof(struct
bevanda_da_preparare) * mysql_stmt_num_rows(visualizza_bevande_da_preparare));
    if(listaBevandeDaPreparare == NULL)
        goto out;
    memset(listaBevandeDaPreparare, 0, sizeof(*listaBevandeDaPreparare) + sizeof(struct
bevanda_da_preparare) * mysql_stmt_num_rows(visualizza_bevande_da_preparare));
    listaBevandeDaPreparare->num_entries =
mysql_stmt_num_rows(visualizza_bevande_da_preparare);


    set_binding_param(&param[0], MYSQL_TYPE_LONG, &comanda, sizeof(comanda));
    set_binding_param(&param[1], MYSQL_TYPE_LONG, &idBevanda, sizeof(idBevanda));
    set_binding_param(&param[2], MYSQL_TYPE_VAR_STRING, nomeBevanda,
NOME_PRODOTTO_LEN);

    if(mysql_stmt_bind_result(visualizza_bevande_da_preparare, param)) {
        print_stmt_error(visualizza_bevande_da_preparare, "Impossibile effettuare il bind dei parametri
per visualizzare le bevande da preparare\n");
        free(listaBevandeDaPreparare);
        listaBevandeDaPreparare = NULL;
        goto out;
    }

    while (true) {
        status = mysql_stmt_fetch(visualizza_bevande_da_preparare);

        if (status == 1 || status == MYSQL_NO_DATA)
            break;

        listaBevandeDaPreparare->bevandaDaPreparare[row].comanda = comanda;
        listaBevandeDaPreparare->bevandaDaPreparare[row].idBevanda = idBevanda;
        strcpy(listaBevandeDaPreparare->bevandaDaPreparare[row].nomeBevanda, nomeBevanda);

        row++;
```

```
    }
    out:
    mysql_stmt_free_result(visualizza_bevande_da_preparare);
    while(mysql_stmt_next_result(visualizza_bevande_da_preparare) != -1) {}
    mysql_stmt_reset(visualizza_bevande_da_preparare);
    return listaBevandeDaPreparare;
}

bool do_segna_bevanda_pronta(struct bevanda_stato *bevandaPronta)
{
    MYSQL_BIND param[2];

    set_binding_param(&param[0], MYSQL_TYPE_LONG, &bevandaPronta->idBevanda,
sizeof(bevandaPronta->idBevanda));
    set_binding_param(&param[1], MYSQL_TYPE_LONG, &bevandaPronta->comanda,
sizeof(bevandaPronta->comanda));

    if(mysql_stmt_bind_param(segna_bevanda_pronta, param) != 0) {
        print_stmt_error(segna_bevanda_pronta, "Impossibile effettuare il binding dei parametri per
do_segna_bevanda_pronta");
        return false;
    }

    if (mysql_stmt_execute(segna_bevanda_pronta) != 0) {
        print_stmt_error(segna_bevanda_pronta, "Impossibile eseguire la procedura
'segna_bevanda_pronta'");
        return false;
    }
    puts("\nAggiornamento dello stato della bevanda andato a buon fine");


    mysql_stmt_free_result(segna_bevanda_pronta);
    mysql_stmt_reset(segna_bevanda_pronta);
    return true;
}
```

VIEW

login.h

```
#pragma once
#include "../model/db.h"

extern bool ask_for_relogin(void);
extern void view_login(struct credentials *cred);
```

login.c

```
#include <stdio.h>
#include "login.h"
#include "../utils/io.h"

void view_login(struct credentials *cred)
{
   clear_screen();
   puts("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
   puts("|                                                 |");
   puts("|            SISTEMA DI GESTIONE DELLA PIZZERIA             |");
   puts("|                                                 |");
   puts("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
         get_input("Username: ", USERNAME_LEN, cred->username, false);
         get_input("Password: ", PASSWORD_LEN, cred->password, true);
}
bool ask_for_relogin(void)
{
         return yes_or_no("Vuoi effettuare il login con un altro utente?", 's', 'n', false, true);
}
```

manager.h

```
#pragma  once
#include "../model/db.h"

enum actions {
   CREA_TURNO,
   CREA_TURNO_CAMERIERE,
   CREA_TURNO_TAVOLO,
   ASSOCIA_CAMERIERE_TAVOLO,
   TOGLI_CAMERIERE_TAVOLO,
   REGISTRA_CLIENTE,
   INSERISCI_PIZZA_MENU,
   INSERISCI_INGREDIENTE_MENU,
   INSERISCI_CONDIMENTO_PIZZA,
   INSERISCI_BEVANDA_MENU,
   AGGIORNA_QUANTITA_INGREDIENTE,
   AGGIORNA_QUANTITA_BEVANDA,
   STAMPA_SCONTRINO,
   REGISTRA_PAGAMENTO_SCONTRINO,
   VISUALIZZA_ENTRATA_GIORNALIERA,
   VISUALIZZA_ENTRATA_MENSILE,
   INSERISCI_TAVOLO,
   INSERISCI_CAMERIERE,
   CANCELLA_TURNO_CAMERIERE,
   QUIT,
   END_OF_ACTIONS
};


extern int get_manager_action(void);
extern void get_creazione_turno_info(struct turno *turno);
extern void get_creazione_turno_cameriere_info(struct turno_cameriere *turnoCameriere);
extern void get_plus1_creazione_turno_cameriere_info(struct turno_cameriere *turnoCameriere);
extern void get_plus2_creazione_turno_cameriere_info(struct turno_cameriere *turnoCameriere);
extern void get_creazione_turno_tavolo_info(struct turno_tavolo *turnoTavolo);
extern void get_plus1_creazione_turno_tavolo_info(struct turno_tavolo *turnoTavolo);
extern void get_plus2_creazione_turno_tavolo_info(struct turno_tavolo *turnoTavolo);
extern void get_associazione_cameriere_tavolo_info(struct cameriere_tavolo *camerieTavolo);
extern void get_plus_associazione_cameriere_tavolo_info(struct cameriere_tavolo
*camerieTavolo);
extern void get_rimozione_cameriere_tavolo_info(struct cameriere_tavolo *camerieTavolo);
extern void get_cliente_info(struct cliente *cliente);
extern void get_pizza_menu_info(struct  pizza *pizza);
extern void get_ingrediente_menu_info(struct  ingrediente *ingrediente);
```

```
extern void get_pizza_condimento_info(struct condimento *condimento);
extern void get_plus_pizza_condimento_info(struct condimento *condimento);
extern void get_bevanda_menu_info(struct bevanda *bevanda);
extern void get_plus_registrazione_cliente_info(struct cameriere_tavolo *camerieretavolo);
extern void get_quantita_prodotto_info(struct prodotto_quantita *ingredienteQuantita);
extern void get_tavolo_scontrino_info(struct tavolo *tavolo);
extern void get_pagamento_scontrino_info(struct scontrino *scontrino);
extern void get_giorno_info(char giorno[DATE_LEN]);
extern void get_anno_mese_info(struct anno_mese *annoMese);
extern void get_cameriere_info(struct nuovo_cameriere *nuovoCameriere);
extern void get_tavolo_info(struct nuovo_tavolo *nuovoTavolo);
extern void get_cancellazione_turno_cameriere_info(struct turno_cameriere *turnoCameriere);
```

manager.c

```
#include <stdio.h>

#include "manager.h"
#include "../utils/io.h"
#include "../utils/validation.h"

#define NUM_LEN 10

int get_manager_action(void)
{
    char *options[] = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13", "14", "15",
"16", "17", "18", "19", "20"};

    clear_screen();
    puts("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    puts("|                                                                    |");
    puts("|                      SCHERMATA DEL MANAGER                         |");
    puts("|                                                                    |");
    puts("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
    puts("~~~~~~~~~~~~~~~   Che azione desideri effettuare?   ~~~~~~~~~~~~~~~\n");
    puts("1)  Creare un turno e definire la sua fascia oraria");
    puts("2)  Definire in che turno far lavorare un cameriere");
    puts("3)  Definire in che turno usare un tavolo");
    puts("4)  Associare un cameriere ad un tavolo");
    puts("5)  Togliere un tavolo ad un cameriere");
    puts("6)  Registrare un cliente e assegnargli un tavolo");
    puts("7)  Inserire nel menu una nuova pizza");
    puts("8)  Inserire nel menu un nuovo ingrediente da usare come aggiunta e/o condimento");
    puts("9)  Indicare un ingrediente usato per condire una pizza");
    puts("10) Inserire nel menu una nuova bevanda");
    puts("11) Aggiornare la quantita presente in magazzino di un ingrediente");
    puts("12) Aggiornare la quantita presente in magazzino di una bevanda");
    puts("13) Stampare uno scontrino");
    puts("14) Registrare il pagamento di uno scontrino");
    puts("15) Visualizzare le entrate di una giornata");
    puts("16) Visualizzare le entrate di un mese");
    puts("17) Inserisci un nuovo tavolo nel sistema");
    puts("18) Inserisci un nuovo cameriere nel sistema");
    puts("19) Segnalare che un cameriere non e' presente nel suo turno di lavoro");
    puts("20) Quit");

    return multi_choice("\nScegli un'opzione", options, 20);
}
```

```
void get_creazione_turno_info(struct turno *turno)
{
    clear_screen();
    puts("\n~~~~~~~~~~~~~~  Fornisci i dati richiesti per creare un nuovo turno  ~~~~~~~~~~~~~~\n");
    get_input("Inserisci il nome che vuoi dare al turno: ", NOME_TURNO_LEN, turno->nome,
false);
    puts("\n*** Usare il formato [HH] per gli orari ***\n");

    while (true) {
        get_input("Inserisci l'orario di inizio del turno: ", TIME_LEN, turno->oraInizio, false);
        if(validate_time(turno->oraInizio))
            break;
        fprintf(stderr, "Formato non valido, reinserire l'orario\n");
    }

    while (true) {
        get_input("Inserisci l'orario di fine del turno: ", TIME_LEN, turno->oraFine, false);
        if(validate_time(turno->oraFine))
            break;
        fprintf(stderr, "Formato non valido, reinserire l'orario\n");
    }
}

void get_creazione_turno_cameriere_info(struct turno_cameriere *turnoCameriere)
{
    clear_screen();
    puts("\n~~~~~~~  Fornisci i dati richiesti per definire in che turno far lavorare un cameriere
~~~~~\n");
    get_input("Inserisci il nome del cameriere: ", NOME_CAMERIERE_LEN, turnoCameriere-
>nomeC, false);
    get_input("Inserisci il cognome del cameriere: ", COGNOME_CAMERIERE_LEN,
turnoCameriere->cognomeC, false);
    get_input("Inserisci il turno: ", NOME_TURNO_LEN, turnoCameriere->turno, false);

    while (true) {
        get_input("Inserisci la data [YYYY-MM-DD]: ", DATE_LEN, turnoCameriere->data, false);
        if(validate_date(turnoCameriere->data))
            break;
        fprintf(stderr, "Formato non valido, reinserire la data\n");
    }
}

void get_plus1_creazione_turno_cameriere_info(struct turno_cameriere *turnoCameriere)
```

```
{
   clear_screen();
   printf("\n~~~~~~~   Fornisci i dati del cameriere da far lavorare nel turno di '%s' in data %s
~~~~~\n\n", turnoCameriere->turno, turnoCameriere->data);
   get_input("Inserisci il nome del cameriere: ", NOME_CAMERIERE_LEN, turnoCameriere-
>nomeC, false);
   get_input("Inserisci il cognome del cameriere: ", COGNOME_CAMERIERE_LEN,
turnoCameriere->cognomeC, false);

}

void get_plus2_creazione_turno_cameriere_info(struct turno_cameriere *turnoCameriere)
{
   clear_screen();
   printf("\n~~~~~~~   Fornisci i dati richiesti per definire in che turno del giorno '%s' far lavorare un
cameriere   ~~~~~\n\n", turnoCameriere->data);
   get_input("Inserisci il turno: ", NOME_TURNO_LEN, turnoCameriere->turno, false);
   get_input("Inserisci il nome del cameriere: ", NOME_CAMERIERE_LEN, turnoCameriere-
>nomeC, false);
   get_input("Inserisci il cognome del cameriere: ", COGNOME_CAMERIERE_LEN,
turnoCameriere->cognomeC, false);

}

void get_creazione_turno_tavolo_info(struct turno_tavolo *turnoTavolo)
{
   char str[NUM_LEN];
   char *ptr;

   clear_screen();
   puts("\n~~~~~~~~~~~~   Fornisci i dati richiesti per definire in che turno usare un tavolo
~~~~~~~~~~~\n");
   get_input("Inserisci il numero del tavolo: ", NUM_LEN, str, false);
   turnoTavolo->tavolo = strtoul(str, &ptr, 10);      //per ottenere un unsigned int
   get_input("Inserisci il turno: ", NOME_TURNO_LEN, turnoTavolo->turno, false);

   while (true) {
      get_input("Inserisci la data [YYYY-MM-DD]: ", DATE_LEN, turnoTavolo->data, false);
      if(validate_date(turnoTavolo->data))
         break;
      fprintf(stderr, "Formato non valido, reinserire la data\n");
   }
}
```

```
void get_plus1_creazione_turno_tavolo_info(struct turno_tavolo *turnoTavolo)
{
   char str[NUM_LEN];
   char *ptr;

   clear_screen();
   printf("\n~~~~~~~~~~~~  Fornisci i dati del tavolo da utilizzare nel turno di '%s' in data %s
~~~~~~~~~~~\n\n", turnoTavolo->turno, turnoTavolo->data);
   get_input("Inserisci il numero del tavolo: ", NUM_LEN, str, false);
   turnoTavolo->tavolo = strtoul(str, &ptr, 10);
}

void get_plus2_creazione_turno_tavolo_info(struct turno_tavolo *turnoTavolo)
{
   char str[NUM_LEN];
   char *ptr;

   clear_screen();
   printf("\n~~~~~~~~~~~~  Fornisci i dati richiesti per definire in che turno del giorno '%s' utilizzare
un tavolo  ~~~~~~~~~~\n\n", turnoTavolo->data);
   get_input("Inserisci il turno: ", NOME_TURNO_LEN, turnoTavolo->turno, false);
   get_input("Inserisci il numero del tavolo: ", NUM_LEN, str, false);
   turnoTavolo->tavolo = strtoul(str, &ptr, 10);
}

void get_associazione_cameriere_tavolo_info(struct cameriere_tavolo *camerieraTavolo)
{
   char str[NUM_LEN];
   char *ptr;

   clear_screen();
   puts("\n~~~~~~~~~~~~  Fornisci i dati richiesti per associare un cameriere ad un tavolo
~~~~~~~~~~~\n");
   get_input("Inserisci il numero del tavolo: ", NUM_LEN, str, false);
   camerieraTavolo->tavolo = strtoul(str, &ptr, 10);
   get_input("Inserisci il nome del cameriere: ", NOME_CAMERIERE_LEN, camerieraTavolo-
>nomeC, false);
   get_input("Inserisci il cognome del cameriere: ", COGNOME_CAMERIERE_LEN,
camerieraTavolo->cognomeC, false);

}

extern void get_plus_associazione_cameriere_tavolo_info(struct cameriere_tavolo
*camerieraTavolo)
```

```
{
    char str[NUM_LEN];
    char *ptr;

    clear_screen();
    printf("\n~~~~~~~~~~~~  Fornisci i dati richiesti per associare il cameriere '%s %s' ad un altro
tavolo  ~~~~~~~~~~\n\n", camiereTavolo->nomeC, camiereTavolo->cognomeC);
    get_input("Inserisci il numero del tavolo: ", NUM_LEN, str, false);
    camiereTavolo->tavolo = strtoul(str, &ptr, 10);
}

void get_rimozione_cameriere_tavolo_info(struct cameriere_tavolo *camiereTavolo)
{
    char str[NUM_LEN];
    char *ptr;

    clear_screen();
    puts("\n~~~~~~~~~~~~  Fornisci i dati richiesti per togliere un tavolo ad un cameriere
~~~~~~~~~~\n");
    get_input("Inserisci il nome del cameriere: ", NOME_CAMERIERE_LEN, camiereTavolo-
>nomeC, false);
    get_input("Inserisci il cognome del cameriere: ", COGNOME_CAMERIERE_LEN,
camiereTavolo->cognomeC, false);
    get_input("Inserisci il numero del tavolo: ", NUM_LEN, str, false);
    camiereTavolo->tavolo = strtoul(str, &ptr, 10);
}

void get_cliente_info(struct cliente *cliente)
{
    char str[NUM_LEN];
    char *ptr;

    clear_screen();
    puts("\n~~~~~~~~~~~~  Fornisci i dati del cliente da registrare  ~~~~~~~~~~\n");
    get_input("Inserisci il nome del cliente: ", NOME_CLIENTE_LEN, cliente->nome, false);
    get_input("Inserisci il cognome del cliente: ", COGNOME_CLIENTE_LEN, cliente->cognome,
false);
    get_input("Inserisci il numero di commensali: ", NUM_LEN, str, false);
    cliente->numeroCommensali = strtoul(str, &ptr, 10);

}

void get_plus_registrazione_cliente_info(struct cameriere_tavolo *camiereTavolo)
{
```

```
    clear_screen();
    printf("\n~~~~~~~~~~~~  Fornisci i dati del cameriere che desideri associare al tavolo %u
~~~~~~~~~~~\n\n", camiereTavolo->tavolo);
    get_input("Inserisci il nome del cameriere: ", NOME_CAMERIERE_LEN, camiereTavolo-
>nomeC, false);
    get_input("Inserisci il cognome del cameriere: ", COGNOME_CAMERIERE_LEN,
camiereTavolo->cognomeC, false);

}

void get_pizza_menu_info(struct pizza *pizza)
{
    clear_screen();
    puts("\n~~~~~~~~~~~~  Fornisci le informazioni sulla pizza da inserire nel menu  ~~~~~~~~~~\
n");
    get_input("Inserisci il nome della pizza: ", NOME_PRODOTTO_LEN, pizza->nome, false);
    get_input("Inserisci il prezzo della pizza: ", PREZZO_PIZZA_LEN, pizza->prezzo, false);
}

void get_ingrediente_menu_info(struct ingrediente *ingrediente)
{
    clear_screen();
    puts("\n~~~~~~~~~~~~  Fornisci le informazioni sull'ingrediente da inserire nel menu
~~~~~~~~~~~\n");
    get_input("Inserisci il nome dell'ingrediente: ", NOME_PRODOTTO_LEN, ingrediente->nome,
false);
    get_input("Inserisci il prezzo dell'ingrediente: ", PREZZO_PIZZA_LEN, ingrediente->prezzo,
false);
}

void get_pizza_condimento_info(struct condimento *condimento)
{
    clear_screen();
    puts("\n~~~~~~~~~~~~  Fornisci le informazioni sulla pizza e sull'ingrediente da usare come
condimento  ~~~~~~~~~~\n");
    get_input("Inserisci il nome della pizza: ", NOME_PRODOTTO_LEN, condimento->pizza,
false);
    get_input("Inserisci il nome dell'ingrediente: ", NOME_PRODOTTO_LEN, condimento-
>ingrediente, false);
}

void get_plus_pizza_condimento_info(struct condimento *condimento)
{
    clear_screen();
```

```
    printf("\n~~~~~~~~~~~~  Fornisci le informazioni sull'ingrediente da usare come condimento per la pizza %s  ~~~~~~~~~~~\n\n", condimento->pizza);
    get_input("Inserisci il nome dell'ingrediente: ", NOME_PRODOTTO_LEN, condimento->ingrediente, false);

}


void get_bevanda_menu_info(struct bevanda *bevanda)
{
    clear_screen();
    puts("\n~~~~~~~~~~~~  Fornisci le informazioni sulla bevanda da inserire nel menu ~~~~~~~~~~~\n");
    get_input("Inserisci il nome della bevanda: ", NOME_PRODOTTO_LEN, bevanda->nome, false);
    get_input("Inserisci il prezzo della bevanda: ", PREZZO_PIZZA_LEN, bevanda->prezzo, false);
}

void get_quantita_prodotto_info(struct prodotto_quantita *prodottoQuantita)
{
    char str[NUM_LEN];
    char *ptr;

    clear_screen();
    puts("\n~~~~~~~~~~~~  Fornisci i dati richiesti per aggiornare la quantita'  ~~~~~~~~~~\n");
    get_input("Inserisci il nome del prodotto: ", NOME_PRODOTTO_LEN, prodottoQuantita->nome, false);
    get_input("Inserisci la quantita' che vuoi aggiungere: ", NUM_LEN, str, false);
    prodottoQuantita->quantita = strtoul(str, &ptr, 10);
}

void get_tavolo_scontrino_info(struct tavolo *tavolo)
{
    char str[NUM_LEN];
    char *ptr;

    clear_screen();
    puts("\n~~~~~~~~~~~~  Fornisci i dati richiesti per la stampa dello scontrino  ~~~~~~~~~~\n");
    get_input("Inserisci il numero del tavolo: ", NUM_LEN, str, false);
    tavolo->numeroTavolo = strtoul(str, &ptr, 10);
}

void get_pagamento_scontrino_info(struct scontrino *scontrino)
{
```

```
    clear_screen();
    puts("\n~~~~~~~~~~~~   Fornisci i dati richiesti per il pagamento dello scontrino   ~~~~~~~~~~\
n");
    get_input("Inserisci la data e l'ora in cui e' stato stampato lo scontrino [YYYY-MM-DD
HH:MM:SS]: ", DATETIME_LEN, scontrino->id, false);
}

void get_giorno_info(char giorno[DATE_LEN]) {
    clear_screen();
    puts("\n~~~~~~~~~~~~   Fornisci i dati richiesti per la visualizzazione delle entrate giornaliere
~~~~~~~~~~\n");
    while (true) {
        get_input("Inserisci la data [YYYY-MM-DD]: ", DATE_LEN, giorno, false);
        if (validate_date(giorno))
            break;
        fprintf(stderr, "Formato non valido, reinserire la data\n");
    }
}

void get_anno_mese_info(struct anno_mese *annoMese)
{
    char str[NUM_LEN];

    clear_screen();
    puts("\n~~~~~~~~~~~~   Fornisci i dati richiesti per la visualizzazione delle entrate mensili
~~~~~~~~~~\n");
    get_input("Inserisci l'anno [YYYY]: ", NUM_LEN, str, false);
    annoMese->anno = strtol(str, NULL, 10);
    while (true){
            get_input("Inserisci il mese [MM]: ", NUM_LEN, str, false);
            annoMese->mese = strtoul(str, NULL, 10);
            if(annoMese->mese >= 1 && annoMese->mese <= 12)
                break;
            fprintf(stderr, "Formato non valido, reinserire il mese\n");
            }
}


void get_tavolo_info(struct nuovo_tavolo *nuovoTavolo)
{
    char str[NUM_LEN];
    char *ptr;

    clear_screen();
```

```
    puts("\n~~~~~~~~~~~~   Fornisci i dati richiesti per inserire un nuovo tavolo   ~~~~~~~~~~~\n");

    get_input("Inserisci il numero del tavolo: ", NUM_LEN, str, false);
    nuovoTavolo->numeroTavolo = strtoul(str, &ptr, 10);
    get_input("Inserisci il numero di posti che il tavolo mette a disposizione: ", NUM_LEN, str,
false);
    nuovoTavolo->numeroPosti = strtoul(str, &ptr, 10);
}

void get_cameriere_info(struct nuovo_cameriere *nuovoCameriere)
{
    clear_screen();
    puts("\n~~~~~~~~~~~~   Fornisci i dati richiesti per inserire un nuovo cameriere   ~~~~~~~~~~\
n");
    get_input("Inserisci il nome del cameriere: ", NOME_CAMERIERE_LEN, nuovoCameriere-
>nome, false);
    get_input("Inserisci il cognome del cameriere: ", COGNOME_CAMERIERE_LEN,
nuovoCameriere->cognome, false);

}

void get_cancellazione_turno_cameriere_info(struct turno_cameriere *turnoCameriere)
{
    clear_screen();
    puts("\n~~~~~~~~~~~~   Fornisci i dati richiesti per cancellare il turno di un cameriere
~~~~~~~~~~\n");
    get_input("Inserisci il nome del cameriere: ", NOME_CAMERIERE_LEN, turnoCameriere-
>nomeC, false);
    get_input("Inserisci il cognome del cameriere: ", COGNOME_CAMERIERE_LEN,
turnoCameriere->cognomeC, false);
    get_input("Inserisci il turno: ", NOME_TURNO_LEN, turnoCameriere->turno, false);

    while (true) {
        get_input("Inserisci la data [YYYY-MM-DD]: ", DATE_LEN, turnoCameriere->data, false);
        if(validate_date(turnoCameriere->data))
            break;
        fprintf(stderr, "Formato non valido, reinserire la data\n");
    }
}
```

cameriere.h

```
#pragma  once
#include "../model/db.h"

enum actions {
   VISUALIZZA_TAVOLI_OCCUPATI,
   VISUALIZZA_TAVOLI_COMANDE_SERVITE,
   REGISTRA_COMANDA,
   VISUALIZZA_COSA_E_DOVE_SERVIRE,
   ORDINA_PIZZA,
   ORDINA_BEVANDA,
   SEGNA_PIZZA_SERVITA,
   SEGNA_BEVANDA_SERVITA,
   QUIT,
   END_OF_ACTIONS
};

extern int get_cameriere_action(void);
extern int get_ordinazione_info(void);

extern void get_registrazione_comanda_info(struct tavolo *numeroTavolo);
extern void get_bevanda_effettiva_info(struct bevanda_effettiva *bevandaEffettiva);
extern void get_plus_bevanda_effettiva_info(struct bevanda_effettiva *bevandaEffettiva);
extern void get_bevanda_servita_info(struct bevanda_stato *bevandaServita);
extern void get_plus_bevanda_servita_info(struct bevanda_stato *bevandaServita);
extern void get_pizza_effettiva_info(struct pizza_effettiva *pizzaEffettiva);
extern void get_plus_pizza_effettiva_info(struct pizza_effettiva *pizzaEffettiva);
extern void get_pizza_servita_info(struct pizza_stato *pizzaServita);
extern void get_plus_pizza_servita_info(struct pizza_stato *pizzaServita);
extern void get_ingrediente_aggiunta_info(char ingrediente[NOME_PRODOTTO_LEN]);

extern void print_tavoli_occupati(struct lista_tavoli *listaTavoliOccupati);
extern void print_tavoli_comandeServite(struct lista_tavoli *listaTavoliComandeServite);
extern void print_cose_da_servire(struct da_servire *daServire);
```

cameriere.c

```
#include <stdio.h>
#include "cameriere.h"
#include "../utils/io.h"
#include "../utils/validation.h"

#define NUM_LEN 5

int get_cameriere_action(void)
{
    char *options[] = {"1", "2", "3", "4", "5", "6", "7", "8", "9"};

    clear_screen();
    puts("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    puts("|                                                      |");
    puts("|                 SCHERMATA DEL CAMERIERE                  |");
    puts("|                                                      |");
    puts("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
    puts("~~~~~~~~~~~~~~~   Che azione desideri effettuare?   ~~~~~~~~~~~~~~~\n");
    puts("1) Visualizzare quali dei miei tavoli sono occupati");
    puts("2) Visualizzare in quali dei miei tavoli le comande sono state completamente servite");
    puts("3) Registrare una nuova comanda");
    puts("4) Visualizzare cosa devo servire ed in che tavolo");
    puts("5) Aggiungere una pizza ad una comanda");
    puts("6) Aggiungere una bevanda ad una comanda");
    puts("7) Segnare che una pizza e' stata servita");
    puts("8) Segnare che una bevanda e' stata servita");
    puts("9) Quit");

    return multi_choice("\nScegli un'opzione", options, 9);
}

int get_ordinazione_info(void) {
    char *options[] = {"1", "2", "3"};

    clear_screen();
    puts("~~~~~~~~~~~~~~~~~~~~~~~~~~   Che azione desideri effettuare?
~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
    puts("1) Prendere l'ordinazione di una pizza");
    puts("2) Prendere l'ordinazione di una bevanda");
    puts("3) Tornare alla schermata iniziale");

    return multi_choice("\nScegli un'opzione", options, 3);
}
```

```
void get_registrazione_comanda_info(struct tavolo *tavolo)
{
   char str[NUM_LEN];
   char *ptr;

   clear_screen();
   puts("~~~~~~~~  Fornisci i dati richiesta per la registrazione di una comanda  ~~~~~~~~\n");
   get_input("Inserisci il numero del tavolo: ", NUM_LEN, str, false);
   tavolo->numeroTavolo = strtoul(str, &ptr, 10);

}

void get_bevanda_effettiva_info(struct bevanda_effettiva *bevandaEffettiva)
{
   char str[NUM_LEN];
   char *ptr;

   clear_screen();
   puts("~~~~~~~~  Fornisci i dati richiesta per l'ordinazione di una bevanda  ~~~~~~~~\n");
   get_input("Inserisci il numero della comanda: ", NUM_LEN, str, false);
   get_input("Inserisci il nome della bevanda: ", NOME_PRODOTTO_LEN, bevandaEffettiva-
>nomeBevanda, false);
   bevandaEffettiva->comanda = strtoul(str, &ptr, 10);
}

void get_plus_bevanda_effettiva_info(struct bevanda_effettiva *bevandaEffettiva)
{
   clear_screen();
   printf("~~~~~~~~  Fornisci i dati della bevanda da aggiungere alla comanda %u  ~~~~~~~~\n\n",
bevandaEffettiva->comanda);
   get_input("Inserisci il nome della bevanda: ", NOME_PRODOTTO_LEN, bevandaEffettiva-
>nomeBevanda, false);
}

void get_bevanda_servita_info(struct bevanda_stato *bevandaServita)
{
   char str[NUM_LEN];
   char *ptr;

   clear_screen();
   puts("~~~~~~~~  Fornisci i dati della bevanda da segnare come servita  ~~~~~~~~\n");
   get_input("Inserisci l'id della bevanda: ", NUM_LEN, str, false);
   bevandaServita->idBevanda = strtoul(str, &ptr, 10);
```

```
    get_input("Inserisci il numero della comanda: ", NUM_LEN, str, false);
    bevandaServita->comanda = strtoul(str,&ptr, 10);
}

void get_plus_bevanda_servita_info(struct bevanda_stato *bevandaServita)
{
    char str[NUM_LEN];
    char *ptr;

    clear_screen();
    printf("~~~~~~~~~  Fornisci i dati della bevanda appartenente alla comanda %u da segnare come
servita  ~~~~~~~~~\n\n", bevandaServita->comanda);
    get_input("Inserisci l'id della bevanda: ", NUM_LEN, str, false);
    bevandaServita->idBevanda = strtoul(str, &ptr, 10);
}

void get_pizza_effettiva_info(struct pizza_effettiva *pizzaEffettiva)
{
    char str[NUM_LEN];
    char *ptr;

    clear_screen();
    puts("~~~~~~~~~  Fornisci i dati richiesta per l'ordinazione di una pizza  ~~~~~~~~~\n");
    get_input("Inserisci il numero della comanda: ", NUM_LEN, str, false);
    pizzaEffettiva->comanda = strtoul(str, &ptr, 10);
    get_input("Inserisci il nome della pizza: ", NOME_PRODOTTO_LEN, pizzaEffettiva-
>nomePizza, false);
}

void get_plus_pizza_effettiva_info(struct pizza_effettiva *pizzaEffettiva)
{
    clear_screen();
    printf("~~~~~~~~~~~  Fornisci i dati della pizza da aggiungere alla comanda %u  ~~~~~~~~~~~\n\
n", pizzaEffettiva->comanda);
    get_input("Inserisci il nome della pizza: ", NOME_PRODOTTO_LEN, pizzaEffettiva-
>nomePizza, false);
}

void get_pizza_servita_info(struct pizza_stato *pizzaServita)
{
    char str[NUM_LEN];
    char *ptr;

    clear_screen();
```

```
    puts("~~~~~~~~  Fornisci i dati della pizza da segnare come servita  ~~~~~~~~\n");
    get_input("Inserisci l'id della pizza: ", NUM_LEN, str, false);
    pizzaServita->idPizza = strtoul(str, &ptr, 10);
    get_input("Inserisci il numero della comanda: ", NUM_LEN, str, false);
    pizzaServita->comanda = strtoul(str,&ptr, 10);
}

void get_plus_pizza_servita_info(struct pizza_stato *pizzaServita)
{
    char str[NUM_LEN];
    char *ptr;

    clear_screen();
    printf("~~~~~~~~  Fornisci i dati della pizza appartenente alla comanda %u da segnare come
servita  ~~~~~~~~\n\n", pizzaServita->comanda);
    get_input("Inserisci l'id della pizza: ", NUM_LEN, str, false);
    pizzaServita->idPizza = strtoul(str, &ptr, 10);
}

void get_ingrediente_aggiunta_info(char ingrediente[NOME_PRODOTTO_LEN])
{
    puts("\n\n\n~~~~~~~~   Fornisci i dati dell'ingrediente da usare come aggiunta   ~~~~~~~\n");
    get_input("Inserisci il nome dell'ingrediente: ", NOME_PRODOTTO_LEN, ingrediente, false);
}

void print_tavoli_occupati(struct lista_tavoli *listaTavoliOccupati)
{
    clear_screen();
    puts("~~~~~~~~~~~~~~~   Tavoli occupati   ~~~~~~~~~~~~~~~\n");

    for (size_t i = 0; i < listaTavoliOccupati->num_entries; ++i) {
        printf("Numero tavolo:  %u\n", listaTavoliOccupati->tavolo[i].numeroTavolo);
    }
}

void print_tavoli_comandeServite(struct lista_tavoli *listaTavoliComandeServite)
{
    clear_screen();
    puts("~~~~~~~~~~~~~~~   Tavoli in cui le comande sono state completamente servite
~~~~~~~~~~~~~~~\n");

    for (size_t i = 0; i < listaTavoliComandeServite->num_entries; i++) {
        printf("Numero tavolo:  %u\n", listaTavoliComandeServite->tavolo[i].numeroTavolo);
    }
```

```
}


void print_cose_da_servire(struct da_servire *daServire)
{
   clear_screen();
   puts("~~~~~~~~~~~~~~~~~   Prodotti da servire   ~~~~~~~~~~~~~~~~");
   size_t j;

   for (size_t i = 0; i < daServire->num_comande; i++) {
      puts("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
      printf("SERVIRE AL TAVOLO: %u\t(comanda: %u)\n", daServire-
>comandeDaServire[i].tavolo, daServire->comandeDaServire[i].comanda);
      puts("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
      j = 0;
      while (j + 1 < daServire->comandeDaServire[i].num_prodotti && daServire-
>comandeDaServire[i].prodottiDaServire[j].id < daServire-
>comandeDaServire[i].prodottiDaServire[j+1].id) {
         printf("\t~~ %s\t(id: %u)\n", daServire->comandeDaServire[i].prodottiDaServire[j].nome,
daServire->comandeDaServire[i].prodottiDaServire[j].id);
         j++;
      }
      printf("\t~~ %s\t(id: %u)\n", daServire-
>comandeDaServire[i].prodottiDaServire[j].nome,daServire-
>comandeDaServire[i].prodottiDaServire[j].id);
      j++;

      if(j < daServire->comandeDaServire[i].num_prodotti)
         puts("\t_____\n");
      while (j  < daServire->comandeDaServire[i].num_prodotti) {
         printf("\t~~ %s\t(id: %u)\n", daServire->comandeDaServire[i].prodottiDaServire[j].nome,
daServire->comandeDaServire[i].prodottiDaServire[j].id);
         j++;
      }
      puts("\n");
   }
}
```

pizzaiolo.h

```
#pragma once
#include "../model/db.h"

enum actions {
   VISUALIZZA_PIZZE_DA_PREPARARE, //e prendi in carico
   SEGNA_PIZZA_PRONTA,
   QUIT,
   END_OF_ACTIONS
};

extern int get_pizzaiolo_action(void);
extern void get_pizza_pronta_info(struct pizza_stato *pizzaPronta);
extern void get_plus_pizza_pronta_info(struct pizza_stato *pizzaPronta);
extern void print_pizze_da_preparare(struct pizze_da_preparare *pizzeDaPreparare);
```

pizzaiolo.c

```
#include <stdio.h>
#include "pizzaiolo.h"
#include "../utils/io.h"
#include "../utils/validation.h"

#define NUM_LEN 5

int get_pizzaiolo_action(void)
{
   char *options[] = {"1", "2", "3"};

   clear_screen();
   puts("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
   puts("|                                                       |");
   puts("|                 SCHERMATA DEL PIZZAIOLO                 |");
   puts("|                                                       |");
   puts("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
   puts("~~~~~~~~~~~~~~~~~   Che azione desideri effettuare?   ~~~~~~~~~~~~~~~~~\n");
   puts("1) Visualizzare le pizze da preparare e prenderle in carico");
   puts("2) Segnare che una pizza e' pronta per essere servita");
   puts("3) Quit");

   return multi_choice("\nScegli un'opzione", options, 3);
}

void get_pizza_pronta_info(struct pizza_stato *pizzaPronta)
{
   char str[NUM_LEN];
   char *ptr;

   clear_screen();
   puts("~~~~~~~~~~  Fornisci i dati della pizza da segnare come pronta  ~~~~~~~~~~\n");
   get_input("Inserisci l'id della pizza: ", NUM_LEN, str, false);
   pizzaPronta->idPizza = strtoul(str, &ptr, 10);
   get_input("Inserisci il numero della comanda: ", NUM_LEN, str, false);
   pizzaPronta->comanda = strtoul(str,&ptr, 10);
}

void get_plus_pizza_pronta_info(struct pizza_stato *pizzaPronta)
{
   char str[NUM_LEN];
   char *ptr;
```

```
    clear_screen();
    printf("~~~~~~~~  Fornisci i dati della pizza appartenente alla comanda %u da segnare come
pronta  ~~~~~~~~\n\n", pizzaPronta->comanda);
    get_input("Inserisci l'id della pizza: ", NUM_LEN, str, false);
    pizzaPronta->idPizza = strtoul(str, &ptr, 10);
}


void print_pizze_da_preparare(struct pizze_da_preparare *pizzeDaPreparare)
{
    clear_screen();
    puts("~~~~~~~~~~~~~~~~   Pizze da preparare   ~~~~~~~~~~~~~~~");

    for (size_t i = 0; i < pizzeDaPreparare->num_pizze; i++) {
        printf("\n\nPIZZA: %s\t(comanda: %u,  id: %u)\n", pizzeDaPreparare->pizzaInfo[i].nomePizza,
pizzeDaPreparare->pizzaInfo[i].comanda, pizzeDaPreparare->pizzaInfo[i].idPizza);

        for (size_t j = 0; j < pizzeDaPreparare->pizzaInfo[i].num_aggiunte; j++) {
            printf("\t~~ AGGIUNTA: %s\n", pizzeDaPreparare->pizzaInfo[i].aggiunte[j].ingrediente);
        }
        puts("_____________________________________________________");
    }
}
```

barista.h

```
#pragma  once
#include "../model/db.h"

enum actions {
   VISUALIZZA_BEVANDE_DA_PREPARARE, //e prendi in carico
   SEGNA_BEVANDA_PRONTA,
   QUIT,
   END_OF_ACTIONS
};

extern int get_barista_action(void);

extern void get_bevanda_pronta_info(struct bevanda_stato *bevandaPronta);
extern void get_plus_bevanda_pronta_info(struct bevanda_stato *bevandaPronta);

extern void print_bevande_da_preparare(struct lista_bevande_da_preparare
*listaBevandeDaPreparare);
```

barista.c

```
#include <stdio.h>

#include "barista.h"
#include "../utils/io.h"
#include "../utils/validation.h"

#define NUM_LEN 5

int get_barista_action(void)
{
   char *options[] = {"1", "2", "3"};

   clear_screen();
   puts("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
   puts("|                                                  |");
   puts("|                  SCHERMATA DEL BARISTA                  |");
   puts("|                                                  |");
   puts("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
   puts("~~~~~~~~~~~~~~~   Che azione desideri effettuare?   ~~~~~~~~~~~~~~~\n");
   puts("1) Visualizzare le bevande da preparare e prenderle in carico");
   puts("2) Segnare che una bevanda e' pronta per essere servita");
   puts("3) Quit");

   return multi_choice("\nScegli un'opzione", options, 3);
}

void get_bevanda_pronta_info(struct bevanda_stato *bevandaPronta)
{
   char str[NUM_LEN];
   char *ptr;

   clear_screen();
   puts("~~~~~~~~~  Fornisci i dati della bevanda da segnare come pronta  ~~~~~~~~~\n");
   get_input("Inserisci l'id della bevanda: ", NUM_LEN, str, false);
   bevandaPronta->idBevanda = strtoul(str, &ptr, 10);
   get_input("Inserisci il numero della comanda: ", NUM_LEN, str, false);
   bevandaPronta->comanda = strtoul(str,&ptr, 10);
}

void get_plus_bevanda_pronta_info(struct bevanda_stato *bevandaPronta)
{
   char str[NUM_LEN];
   char *ptr;
```

```
    clear_screen();
    printf("~~~~~~~~  Fornisci i dati della bevanda appartenente alla comanda %u da segnare come
pronta  ~~~~~~~~\n\n", bevandaPronta->comanda);
    get_input("Inserisci l'id della bevanda: ", NUM_LEN, str, false);
    bevandaPronta->idBevanda = strtoul(str, &ptr, 10);

}

void print_bevande_da_preparare(struct lista_bevande_da_preparare *listaBevandeDaPreparare)
{
    clear_screen();
    puts("~~~~~~~~~~~~~~~   Bevande da preparare   ~~~~~~~~~~~~~~~\n");
    for (size_t i = 0; i < listaBevandeDaPreparare->num_entries; i++) {
        printf("\n~~ BEVANDA: %s\t(comanda: %u,  id: %u)\n", listaBevandeDaPreparare-
>bevandaDaPreparare[i].nomeBevanda, listaBevandeDaPreparare-
>bevandaDaPreparare[i].comanda, listaBevandeDaPreparare→bevandaDaPreparare[i].idBevanda);
    }
}
```

CONTROLLER

login.h

```
#pragma once
#include <stdbool.h>


extern bool login(void);
```

login.c

```
#include <stdbool.h>

#include "login.h"
#include "manager.h"
#include "cameriere.h"
#include "pizzaiolo.h"
#include "barista.h"
#include "../view/login.h"


bool login(void)
{
	struct credentials cred;
	view_login(&cred);
	role_t role = attempt_login(&cred);

	switch(role) {
		case MANAGER:
	manager_controller();
			break;
		case CAMERIERE:
	cameriere_controller(&cred);
			break;
      case PIZZAIOLO:
	pizzaiolo_controller();
	break;
      case BARISTA:
	barista_controller();
	break;
		default:
			return false;
	}
	return true;
}
```

manager.h

```
#pragma once

extern void manager_controller(void);
```

manager.c

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

#include "manager.h"
#include "../model/db.h"
#include "../view/manager.h"
#include "../utils/io.h"

static bool crea_turno(void)
{
   struct turno turno;
   memset(&turno, 0, sizeof(turno));
   get_creazione_turno_info(&turno);
   do_crea_turno(&turno);
   return false;
}

static bool plus_crea_turno_cameriere(struct turno_cameriere turnoCameriere)
{
   bool go;
   while (true) {
      go = yes_or_no("\n\nDesideri far lavorare un altro cameriere in questo turno e in questa data?",
's', 'n', false, true);
      if (go) {
         get_plus1_creazione_turno_cameriere_info(&turnoCameriere);
         do_crea_turno_cameriere(&turnoCameriere);
      } else {
         go = yes_or_no("\n\nDesideri definire quali camerieri far lavorare in un altro turno di questo
giorno?",
                        's', 'n', false, true);
         if(go){
            get_plus2_creazione_turno_cameriere_info(&turnoCameriere);
            do_crea_turno_cameriere(&turnoCameriere);
         } else
            return false;
      }
   }
```

```
}

static bool crea_turno_cameriere(void)
{
   struct turno_cameriere turnoCameriere;
   memset(&turnoCameriere, 0, sizeof(turnoCameriere));
   get_creazione_turno_cameriere_info(&turnoCameriere);
   bool done = do_crea_turno_cameriere(&turnoCameriere);
   if(done)
      plus_crea_turno_cameriere(turnoCameriere);
   return false;
}

static bool plus_crea_turno_tavolo(struct turno_tavolo turnoTavolo)
{
   while (true) {
      bool go = yes_or_no("\n\nDesideri utilizzare un altro tavolo in questo turno e in questa data?",
's', 'n', false, true);
      if (go) {
         get_plus1_creazione_turno_tavolo_info(&turnoTavolo);
         do_crea_turno_tavolo(&turnoTavolo);
      } else {
         go = yes_or_no("\n\nDesideri definire quali tavoli utilizzare in un altro turno di questo
giorno?",
                        's', 'n', false, true);
         if(go){
            get_plus2_creazione_turno_tavolo_info(&turnoTavolo);
            do_crea_turno_tavolo(&turnoTavolo);
         } else
            return false;
      }
   }
}

static bool crea_turno_tavolo(void)
{
   struct turno_tavolo turnoTavolo;
   memset(&turnoTavolo, 0, sizeof(turnoTavolo));
   get_creazione_turno_tavolo_info(&turnoTavolo);
   bool done = do_crea_turno_tavolo(&turnoTavolo);
   if(done)
      plus_crea_turno_tavolo(turnoTavolo);
   return false;
}
```

```
static bool plus_associa_cameriere_tavolo(struct cameriere_tavolo camerieреTavolo)
{
   while (true) {
      bool go = yes_or_no("\n\nDesideri associare questo cameriere ad altri tavoli?", 's', 'n', false,
true);
      if (go) {
         get_plus_associazione_cameriere_tavolo_info(&camerieреTavolo);
         do_associa_cameriere_tavolo(&camerieреTavolo);
      } else
         return false;
   }
}

static bool associa_cameriere_tavolo(void)
{
   struct cameriere_tavolo camerieреTavolo;
   memset(&camerieреTavolo, 0, sizeof(camerieреTavolo));
   get_associazione_cameriere_tavolo_info(&camerieреTavolo);
   bool done = do_associa_cameriere_tavolo(&camerieреTavolo);
   if(done)
      plus_associa_cameriere_tavolo(camerieреTavolo);
   return false;
}

static bool togli_cameriere_tavolo(void)
{
   struct cameriere_tavolo camerieреTavolo;
   memset(&camerieреTavolo, 0, sizeof(camerieреTavolo));
   get_rimozione_cameriere_tavolo_info(&camerieреTavolo);
   do_togli_cameriere_tavolo(&camerieреTavolo);
   return false;
}

static bool registra_cliente(void)
{
   int numeroTavolo;
   struct cliente cliente;
   memset(&cliente, 0, sizeof(cliente));
   get_cliente_info(&cliente);
   numeroTavolo = do_registra_cliente(&cliente);

   if(numeroTavolo >= 0) {
      printf("\nAl cliente %s %s e' stato assegnato il tavolo %d", cliente.nome, cliente.cognome,
```

```
numeroTavolo);

        bool plus_actions;
        printf("\n\nSe il tavolo %d non ha un cameriere associato il cliente non potra' in seguito ordinare..", numeroTavolo);
        plus_actions = yes_or_no("\nDesideri associare un cameriere a questo tavolo o cambiarlo nel caso in cui fosse gia' associato?\n\t**Puoi farlo anche in un secondo momento\n     ", 's', 'n', false, true);
        if(plus_actions) {
            struct cameriere_tavolo camerieTavolo;
            memset(&camerieTavolo, 0, sizeof(camerieTavolo));
            camerieTavolo.tavolo = numeroTavolo;
            get_plus_registrazione_cliente_info(&camerieTavolo);
            do_associa_cameriere_tavolo(&camerieTavolo);
            return false;
        }
    }
    return false;
}

static bool inserisci_pizza_menu(void)
{
    struct pizza pizza;
    memset(&pizza, 0, sizeof(pizza));
    get_pizza_menu_info(&pizza);
    do_inserisci_pizza_menu(&pizza);

    while (true) {
        bool plus_actions;
        plus_actions = yes_or_no("\nDesideri inserire un'altra pizza nel menu?", 's', 'n', false, true);
        if (plus_actions) {
            get_pizza_menu_info(&pizza);
            do_inserisci_pizza_menu(&pizza);
        }
        else
            return false;
    }
}

static bool inserisci_ingrediente_menu(void)
{
    struct ingrediente ingrediente;
    memset(&ingrediente, 0, sizeof(ingrediente));
    get_ingrediente_menu_info(&ingrediente);
```

```
    do_inserisci_ingrediente_menu(&ingrediente);

    while (true) {
        bool plus_actions;
        plus_actions = yes_or_no("\nDesideri inserire un altro ingrediente nel menu?", 's', 'n', false,
true);
        if (plus_actions) {
            get_ingrediente_menu_info(&ingrediente);
            do_inserisci_ingrediente_menu(&ingrediente);
        }
        else
            return false;
    }
}

static bool plus_inserisci_condimento_pizza(struct condimento condimento)
{
    while (true) {
        bool go = yes_or_no("\n\nDesideri inserire altri condimenti per questa pizza?", 's', 'n', false,
true);
        if (go) {
            get_plus_pizza_condimento_info(&condimento);
            do_inserisci_condimento_pizza(&condimento);
        } else
            return false;
    }
}

static bool inserisci_condimento_pizza(void)
{
    struct condimento condimento;
    memset(&condimento, 0, sizeof(condimento));
    get_pizza_condimento_info(&condimento);
    bool done = do_inserisci_condimento_pizza(&condimento);
    if(done)
        plus_inserisci_condimento_pizza(condimento);
    return false;
}

static bool inserisci_bevanda_menu(void)
{
    struct bevanda bevanda;
    memset(&bevanda, 0, sizeof(bevanda));
    get_bevanda_menu_info(&bevanda);
```

```
    do_inserisci_bevanda_menu(&bevanda);

    while (true) {
        bool plus_actions;
        plus_actions = yes_or_no("\nDesideri inserire un'altra bevanda nel menu?", 's', 'n', false, true);
        if (plus_actions) {
            get_bevanda_menu_info(&bevanda);
            do_inserisci_bevanda_menu(&bevanda);
        }
        else
            return false;
    }
}

static bool aggiorna_quantita_ingrediente(void)
{
    struct prodotto_quantita ingredienteQuantita;
    memset(&ingredienteQuantita, 0, sizeof(ingredienteQuantita));
    get_quantita_prodotto_info(&ingredienteQuantita);
    do_aggiorna_quantita_ingrediente(&ingredienteQuantita);
    return false;
}

static bool aggiorna_quantita_bevanda(void)
{
    struct prodotto_quantita bevandaQuantita;
    memset(&bevandaQuantita, 0, sizeof(bevandaQuantita));
    get_quantita_prodotto_info(&bevandaQuantita);
    do_aggiorna_quantita_bevanda(&bevandaQuantita);
    return false;
}

static bool stampa_scontrino(void)
{
    struct tavolo tavolo;
    memset(&tavolo, 0, sizeof(tavolo));
    get_tavolo_scontrino_info(&tavolo);
    struct scontrino *scontrino = do_stampa_scontrino(&tavolo);

    if(scontrino != NULL) {
        printf("\n[%s] : scontrino del tavolo %u stampato, prezzo da pagare: %s\n", scontrino->id,
tavolo.numeroTavolo, scontrino->prezzo);

        bool plus_actions;
```

```
        plus_actions = yes_or_no("\nDesideri registrare subito il pagamento di questo scontrino?", 's',
'n', false, true);
        if(plus_actions) {
            do_registra_pagamento_scontrino(scontrino);
            return false;
        }
    }
    return false;
}

static bool registra_pagamento_scontrino(void)
{
    struct scontrino scontrino;
    memset(&scontrino, 0, sizeof(scontrino));
    get_pagamento_scontrino_info(&scontrino);
    do_registra_pagamento_scontrino(&scontrino);
    return false;
}

static bool visualizza_entrata_giornaliera(void)
{
    char giorno[DATE_LEN];
    memset(&giorno, 0, DATE_LEN);
    get_giorno_info(giorno);
    struct entrata *entrataGiornaliera = do_visualizza_entrata_giornaliera(giorno);

    if(entrataGiornaliera != NULL)
        printf("\nLe entrate del giorno %s ammontano a: %s\n", giorno, entrataGiornaliera->tot);
    return false;
}

static bool visualizza_entrata_mensile(void)
{
    struct anno_mese annoMese;
    memset(&annoMese, 0, sizeof(annoMese));
    get_anno_mese_info(&annoMese);
    struct entrata *entrataMensile = do_visualizza_entrata_mensile(&annoMese);

    if(entrataMensile != NULL)
        printf("\nLe entrate del mese %u dell'anno %u ammontano a: %s\n", annoMese.mese,
annoMese.anno, entrataMensile->tot);
    return false;
}
```

```
static bool inserisci_tavolo(void)
{
   struct nuovo_tavolo nuovoTavolo;
   memset(&nuovoTavolo, 0, sizeof(nuovoTavolo));
   get_tavolo_info(&nuovoTavolo);
   do_inserisci_tavolo(&nuovoTavolo);

   while (true) {
      bool plus_actions;
      plus_actions = yes_or_no("\nDesideri inserire un altro tavolo nel sistema?", 's', 'n', false, true);
      if (plus_actions) {
         get_tavolo_info(&nuovoTavolo);
         do_inserisci_tavolo(&nuovoTavolo);
      }
      else
         return false;
   }
}

static bool inserisci_cameriere(void)
{
   struct nuovo_cameriere nuovoCameriere;
   memset(&nuovoCameriere, 0, sizeof(nuovoCameriere));
   get_cameriere_info(&nuovoCameriere);
   do_inserisci_cameriere(&nuovoCameriere);

   while (true) {
      bool plus_actions;
      plus_actions = yes_or_no("\nDesidera inserire un altro cameriere nel sistema?", 's', 'n', false,
true);
      if (plus_actions) {
         get_cameriere_info(&nuovoCameriere);
         do_inserisci_cameriere(&nuovoCameriere);
      }
      else
         return false;
   }
}

static bool cancella_turno_cameriere(void)
{
   struct turno_cameriere turnoCameriere;
   memset(&turnoCameriere, 0, sizeof(turnoCameriere));
   get_cancellazione_turno_cameriere_info(&turnoCameriere);
```

```
    do_cancella_turno_cameriere(&turnoCameriere);
    return false;
}

static bool quit(void) {
    return true;
}

static struct {
    enum actions action;
    bool (*control)(void);
} controls[END_OF_ACTIONS] = {
        {.action = CREA_TURNO, .control = crea_turno},
        {.action = CREA_TURNO_CAMERIERE, .control = crea_turno_cameriere},
        {.action = CREA_TURNO_TAVOLO, .control = crea_turno_tavolo},
        {.action = ASSOCIA_CAMERIERE_TAVOLO, .control = associa_cameriere_tavolo},
        {.action = TOGLI_CAMERIERE_TAVOLO, .control = togli_cameriere_tavolo},
        {.action = REGISTRA_CLIENTE, .control = registra_cliente},
        {.action = INSERISCI_PIZZA_MENU, .control = inserisci_pizza_menu},
        {.action = INSERISCI_INGREDIENTE_MENU, .control = inserisci_ingrediente_menu},
        {.action = INSERISCI_CONDIMENTO_PIZZA, .control = inserisci_condimento_pizza},
        {.action = INSERISCI_BEVANDA_MENU, .control = inserisci_bevanda_menu},
        {.action = AGGIORNA_QUANTITA_INGREDIENTE, .control =
aggiorna_quantita_ingrediente},
        {.action = AGGIORNA_QUANTITA_BEVANDA, .control = aggiorna_quantita_bevanda},
        {.action = STAMPA_SCONTRINO, .control = stampa_scontrino},
        {.action = REGISTRA_PAGAMENTO_SCONTRINO, .control =
registra_pagamento_scontrino},
        {.action = VISUALIZZA_ENTRATA_GIORNALIERA, .control =
visualizza_entrata_giornaliera},
        {.action = VISUALIZZA_ENTRATA_MENSILE, .control = visualizza_entrata_mensile},
        {.action = INSERISCI_TAVOLO, .control = inserisci_tavolo},
        {.action = INSERISCI_CAMERIERE, .control = inserisci_cameriere},
        {.action = CANCELLA_TURNO_CAMERIERE, .control = cancella_turno_cameriere},
        {.action = QUIT, .control = quit}
};

void manager_controller(void)
{
    db_switch_to_manager();

    while(true) {
        int action = get_manager_action();
        if (action >= END_OF_ACTIONS) {
```

```
            fprintf(stderr, "Errore: azione non riconosciuta\n");
            continue;
        }
        if (controls[action].control())
            break;

        press_anykey();
    }
}
```

cameriere.h

```
#pragma once
#include "../model/db.h"

extern void cameriere_controller(struct credentials *credentials );
```

cameriere.c

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

#include "cameriere.h"
#include "../view/cameriere.h"
#include "../utils/io.h"

static bool visualizza_tavoli_occupati(char *username)
{
   struct lista_tavoli *tavoliOccupati = do_visualizza_tavoli_occupati(username);
   if(tavoliOccupati != NULL) {
      print_tavoli_occupati(tavoliOccupati);
      free(tavoliOccupati);
   }
   return false;
}

static bool visualizza_tavoli_comande_servite(char *username)
{
   struct lista_tavoli *tavoliComandeServite = do_visualizza_tavoli_comande_servite(username);
   if(tavoliComandeServite != NULL) {
      print_tavoli_comandeServite(tavoliComandeServite);
      free(tavoliComandeServite);
   }

   return false;
}


static bool visualizza_cosa_e_dove_servire(char *username)
{
   struct da_servire *daServire = do_visualizza_cosa_e_dove_servire(username);
   if(daServire != NULL) {
      print_cose_da_servire(daServire);
      free_da_servire(daServire);
   }
```

```
    return false;
}


static bool plus_ordina_bevanda(struct bevanda_effettiva bevandaEffettiva, char *username)
{
    while (true) {
        bool go = yes_or_no("\n\nDesideri aggiungere altre bevande a questa comanda?", 's', 'n', false,
true);
        if (go) {
            get_plus_bevanda_effettiva_info(&bevandaEffettiva);
            do_ordina_bevanda(&bevandaEffettiva, username);
        } else
            return false;
    }
}


static bool ordina_bevanda(char *username)
{
    struct bevanda_effettiva bevandaEffettiva;
    memset(&bevandaEffettiva, 0, sizeof(bevandaEffettiva));
    get_bevanda_effettiva_info(&bevandaEffettiva);
    bool done = do_ordina_bevanda(&bevandaEffettiva, username);
    if(done)
        plus_ordina_bevanda(bevandaEffettiva, username);
    return false;
}


static bool completa_ordinazione_pizza(struct pizza_effettiva *pizzaEffettiva, char *username)
{
    int idPizza = do_ordina_pizza(pizzaEffettiva, username);
    bool done, go, aggiunte;

    if(idPizza > 0) {
        printf("\nL'id della pizza appena ordinata e': %d\n", idPizza);
        struct pizza_stato pizzaOrdinata;
        memset(&pizzaOrdinata, 0, sizeof(pizzaOrdinata));
        pizzaOrdinata.idPizza = idPizza;
        pizzaOrdinata.comanda = pizzaEffettiva->comanda;

        richiesta_aggiunte:
        aggiunte = yes_or_no(
```

```
            "\n\n~~~~~~~~~~~~~~~~~  Desideri effettuare aggiunte su questa pizza?
~~~~~~~~~~~~~~~~~~\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  ",
            's', 'n', false, true);
      if (aggiunte) {
         struct aggiunta_pizza aggiunta;
         memset(&aggiunta, 0, sizeof(aggiunta));
         while (go) {
            aggiunta.comanda = pizzaEffettiva->comanda;
            aggiunta.idPizza = idPizza;
            get_ingrediente_aggiunta_info(aggiunta.ingrediente);
            do_inserisci_aggiunta_pizza(&aggiunta);

            bool plus_aggiunte = yes_or_no("\nCi sono altre aggiunte da effettuare?", 's', 'n', false,
true);
            if (!plus_aggiunte) {
               done = yes_or_no("\nConfermi che l'ordinazione di questa pizza e' stata completata?",
's', 'n',
                           true, true);
               if (done) {
                  go = false;
                  do_segna_pizza_ordinata(&pizzaOrdinata);
               }
            }
         }
      } else {
         done = yes_or_no("\nConfermi che l'ordinazione di questa pizza e' stata completata?", 's', 'n',
                     true, true);
         if (done)
            do_segna_pizza_ordinata(&pizzaOrdinata);
         else {
            go = true;
            goto richiesta_aggiunte;
         }
      }
      return true;

   } else
      return false;
}

static bool plus_ordina_pizza(struct pizza_effettiva pizzaEffettiva, char *username)
{
   while (true) {
      bool go = yes_or_no("\n\nDesideri aggiungere altre pizze a questa comanda?", 's', 'n', false,
```

```
true);
        if (go) {
            get_plus_pizza_effettiva_info(&pizzaEffettiva);
            completa_ordinazione_pizza(&pizzaEffettiva, username);
        }else
            return false;
    }
}

static bool ordina_pizza(char *username)
{
    struct pizza_effettiva pizzaEffettiva;
    memset(&pizzaEffettiva, 0, sizeof(pizzaEffettiva));
    get_pizza_effettiva_info(&pizzaEffettiva);
    bool done = completa_ordinazione_pizza(&pizzaEffettiva, username);

    if(done)
        plus_ordina_pizza(pizzaEffettiva, username);

    return false;
}


static bool registra_comanda(char *username)
{
    int numeroComanda;
    struct tavolo tavolo;
    memset(&tavolo, 0, sizeof (tavolo));
    get_registrazione_comanda_info(&tavolo);
    numeroComanda = do_registra_comanda(&tavolo, username);

    if(numeroComanda > 0) {
        printf("\nE' stata effettuata la registrazione della comanda con il numero: %d\n\n",
numeroComanda);
        press_anykey();
        while (true) {
            int action = get_ordinazione_info();

            if (action == 0)
            {
                struct pizza_effettiva pizzaEffettiva;
                memset(&pizzaEffettiva, 0, sizeof(pizzaEffettiva));
                pizzaEffettiva.comanda = numeroComanda;
                get_plus_pizza_effettiva_info(&pizzaEffettiva);
```

```
            completa_ordinazione_pizza(&pizzaEffettiva, username);
            plus_ordina_pizza(pizzaEffettiva, username);
        }
        else if (action == 1)
        {
            struct bevanda_effettiva bevandaEffettiva;
            memset(&bevandaEffettiva, 0, sizeof(bevandaEffettiva));
            bevandaEffettiva.comanda = numeroComanda;
            get_plus_bevanda_effettiva_info(&bevandaEffettiva);
            bool done = do_ordina_bevanda(&bevandaEffettiva, username);
            if (done)
                plus_ordina_bevanda(bevandaEffettiva, username);
        } else
            return false;
    }
  }
  return false;
}


static bool segna_bevanda_servita(char *username)
{
  bool done;
  struct bevanda_stato bevandaServita;
  memset(&bevandaServita, 0, sizeof(bevandaServita));
  get_bevanda_servita_info(&bevandaServita);
  done = do_segna_bevanda_servita(&bevandaServita, username);

  bool plus_actions;
  while (done) {
    plus_actions = yes_or_no("\nDesideri segnare come servite altre bevande di questa comanda?",
's', 'n', false, true);
    if (plus_actions) {
      get_plus_bevanda_servita_info(&bevandaServita);
      done = do_segna_bevanda_servita(&bevandaServita, username);
    }else
      return false;
  }
  return false;
}

static bool segna_pizza_servita(char *username)
{
  bool done;
```

```
    struct pizza_stato pizzaServita;
    memset(&pizzaServita, 0, sizeof(pizzaServita));
    get_pizza_servita_info(&pizzaServita);
    done = do_segna_pizza_servita(&pizzaServita, username);

    bool plus_actions;
    while (done) {
        plus_actions = yes_or_no("\nDesideri segnare come servite altre pizze di questa comanda?", 's',
'n', false, true);
        if (plus_actions) {
            get_plus_pizza_servita_info(&pizzaServita);
            done = do_segna_pizza_servita(&pizzaServita, username);
        }else
            return false;
    }
    return false;
}

static bool quit(char *username) {
    return true;
}

static struct {
    enum actions action;
    bool (*control)(char *username);
} controls[END_OF_ACTIONS] = {
        {.action = VISUALIZZA_TAVOLI_OCCUPATI, .control = visualizza_tavoli_occupati},
        {.action = VISUALIZZA_TAVOLI_COMANDE_SERVITE, .control =
visualizza_tavoli_comande_servite},
        {.action = REGISTRA_COMANDA, .control = registra_comanda},
        {.action = VISUALIZZA_COSA_E_DOVE_SERVIRE, .control =
visualizza_cosa_e_dove_servire},
        {.action = ORDINA_PIZZA, .control = ordina_pizza},
        {.action = ORDINA_BEVANDA, .control = ordina_bevanda},
        {.action = SEGNA_PIZZA_SERVITA, .control = segna_pizza_servita},
        {.action = SEGNA_BEVANDA_SERVITA, .control = segna_bevanda_servita},
        {.action = QUIT, .control = quit}
};


void cameriere_controller(struct credentials *credentials)
{
    db_switch_to_cameriere();
    while (true) {
```

```
        int action = get_cameriere_action();
        if(action >= END_OF_ACTIONS) {
            fprintf(stderr, "Errore: azione non riconosciuta\n");
            continue;
        }
        if (controls[action].control(credentials->username))
            break;
        press_anykey();
    }
}
```

pizzaiolo.h

```
#pragma once

extern void pizzaiolo_controller(void);
```

pizzaiolo.c

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

#include "pizzaiolo.h"
#include "../model/db.h"
#include "../view/pizzaiolo.h"
#include "../utils/io.h"

static bool visualizza_pizze_da_preparare(void)
{
    struct pizze_da_preparare *pizzeDaPreparare = do_visualizza_pizze_da_preparare();
    if(pizzeDaPreparare != NULL) {
        print_pizze_da_preparare(pizzeDaPreparare);
        free_pizze_da_preparare(pizzeDaPreparare);
    }
    return false;
}

static bool segna_pizza_pronta(void)
{
    bool done;
    struct pizza_stato pizzaPronta;
    memset(&pizzaPronta, 0, sizeof(pizzaPronta));
    get_pizza_pronta_info(&pizzaPronta);
    done = do_segna_pizza_pronta(&pizzaPronta);

    bool plus_actions;
    while (done) {
        plus_actions = yes_or_no("\nDesideri segnare come pronte altre pizze di questa comanda?", 's',
'n', false, true);
        if (plus_actions) {
            get_plus_pizza_pronta_info(&pizzaPronta);
            done = do_segna_pizza_pronta(&pizzaPronta);
        }else
            return false;
    }
    return false;
```

```
}

static bool quit(void) {
   return true;
}

static struct {
   enum actions action;
   bool (*control)(void);
} controls[END_OF_ACTIONS] = {
      {.action = VISUALIZZA_PIZZE_DA_PREPARARE, .control =
visualizza_pizze_da_preparare},
      {.action = SEGNA_PIZZA_PRONTA, .control = segna_pizza_pronta},
      {.action = QUIT, .control = quit},
};

void pizzaiolo_controller(void)
{
   db_switch_to_pizzaiolo();

   while (true) {
      int action = get_pizzaiolo_action();
      if(action >= END_OF_ACTIONS) {
         fprintf(stderr, "Errore: azione non riconosciuta\n");
         continue;
      }
      if (controls[action].control())
         break;

      press_anykey();
   }
}
```

barista.h

```
#pragma once

extern void barista_controller(void);
```

barista.c

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

#include "barista.h"
#include "../model/db.h"
#include "../view/barista.h"
#include "../utils/io.h"

static bool visualizza_bevande_da_preparare(void)
{
   struct lista_bevande_da_preparare *bevandeDaPreparare =
do_visualizza_bevande_da_preparare();
   if(bevandeDaPreparare != NULL) {
      print_bevande_da_preparare(bevandeDaPreparare);
      free(bevandeDaPreparare);
   }
   return false;
}

static bool segna_bevanda_pronta(void)
{

   bool done;
   struct bevanda_stato bevandaPronta;
   memset(&bevandaPronta, 0, sizeof(bevandaPronta));
   get_bevanda_pronta_info(&bevandaPronta);
   done = do_segna_bevanda_pronta(&bevandaPronta);

   bool plus_actions;
   while (done) {
      plus_actions = yes_or_no("\nDesideri segnare come servite altre bevande di questa comanda?",
's', 'n', false, true);
      if (plus_actions) {
         get_plus_bevanda_pronta_info(&bevandaPronta);
         done = do_segna_bevanda_pronta(&bevandaPronta);
      }else
         return false;
```

```
    }
    return false;
}

static bool quit(void) {
    return true;
}

static struct {
    enum actions action;
    bool (*control)(void);
} controls[END_OF_ACTIONS] = {
        {.action = VISUALIZZA_BEVANDE_DA_PREPARARE, .control =
visualizza_bevande_da_preparare},
        {.action = SEGNA_BEVANDA_PRONTA, .control = segna_bevanda_pronta},
        {.action = QUIT, .control = quit},
};


void barista_controller(void)
{
    db_switch_to_barista();

    while (true) {
        int action = get_barista_action();
        if(action >= END_OF_ACTIONS) {
            fprintf(stderr, "Errore: azione non riconosciuta\n");
            continue;
        }
        if (controls[action].control())
            break;
        press_anykey();
    }
}
```

UTILS

db.h

```
#pragma once
#include <stdbool.h>
#include <mysql.h>

extern void print_stmt_error (MYSQL_STMT *stmt, char *message);
extern void print_error(MYSQL *conn, char *message);
extern bool setup_prepared_stmt(MYSQL_STMT **stmt, char *statement, MYSQL *conn);
extern void finish_with_error(MYSQL *conn, char *message);
extern void finish_with_stmt_error(MYSQL *conn, MYSQL_STMT *stmt, char *message, bool
close_stmt);
extern void set_binding_param(MYSQL_BIND *param, enum enum_field_types type, void
*buffer, unsigned long len);
extern void date_to_mysql_time(char *str, MYSQL_TIME *time);
extern void time_to_mysql_time(char *str, MYSQL_TIME *time);
extern void init_mysql_timestamp(MYSQL_TIME *time);
extern void mysql_timestamp_to_string(MYSQL_TIME *time, char *str);
extern void mysql_date_to_string(MYSQL_TIME *date, char *str);
extern void datetime_to_mysql_time(char *str, MYSQL_TIME *time);
```

db.c

```
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <mysql.h>

#include "db.h"
#include "../model/db.h"

void print_stmt_error (MYSQL_STMT *stmt, char *message)
{
	fprintf (stderr, "%s\n", message);
	if (stmt != NULL) {
		fprintf (stderr, "Error %u (%s): %s\n",
			mysql_stmt_errno (stmt),
			mysql_stmt_sqlstate(stmt),
			mysql_stmt_error (stmt));
	}
}

void print_error(MYSQL *conn, char *message)
{
	fprintf (stderr, "%s\n", message);
	if (conn != NULL) {
		fprintf (stderr, "Error %u (%s): %s\n",
			mysql_errno (conn), mysql_sqlstate(conn), mysql_error (conn));
	}
}
bool setup_prepared_stmt(MYSQL_STMT **stmt, char *statement, MYSQL *conn)
{
	bool update_length = true;

	*stmt = mysql_stmt_init(conn);
	if (*stmt == NULL)
	{
		print_error(conn, "Could not initialize statement handler");
		return false;
	}

	if (mysql_stmt_prepare (*stmt, statement, strlen(statement)) != 0) {
		print_stmt_error(*stmt, "Could not prepare statement");
		return false;
	}
```

```
	mysql_stmt_attr_set(*stmt, STMT_ATTR_UPDATE_MAX_LENGTH, &update_length);

	return true;
}


void finish_with_error(MYSQL *conn, char *message)
{
	print_error(conn, message);
	mysql_close(conn);
	exit(EXIT_FAILURE);
}


void finish_with_stmt_error(MYSQL *conn, MYSQL_STMT *stmt, char *message, bool
close_stmt)
{
	print_stmt_error(stmt, message);
	if(close_stmt)
		mysql_stmt_close(stmt);
	mysql_close(conn);
	exit(EXIT_FAILURE);
}


void set_binding_param(MYSQL_BIND *param, enum enum_field_types type, void *buffer,
unsigned long len)
{
	memset(param, 0, sizeof(*param));

	param->buffer_type = type;
	param->buffer = buffer;
	param->buffer_length = len;
}


void date_to_mysql_time(char *str, MYSQL_TIME *time)
{
	memset(time, 0, sizeof(*time));
	sscanf(str, "%4d-%2d-%2d", &time->year, &time->month, &time->day);
	time->time_type = MYSQL_TIMESTAMP_DATE;
}
```

```
void time_to_mysql_time(char *str, MYSQL_TIME *time)
{
        memset(time, 0, sizeof(*time));
        sscanf(str, "%02d", &time->hour);
        time->time_type = MYSQL_TIMESTAMP_TIME;
}

void datetime_to_mysql_time(char *str, MYSQL_TIME *time)
{
   memset(time, 0, sizeof(*time));
   sscanf(str, "%4d-%02d-%02d %02d:%02d:%02d", &time->year, &time->month, &time->day,
&time->hour, &time->minute, &time->second);
   time->time_type = MYSQL_TIMESTAMP_DATETIME;
}


void init_mysql_timestamp(MYSQL_TIME *time)
{
        memset(time, 0, sizeof (*time));
        time->time_type = MYSQL_TIMESTAMP_DATETIME;
}

void mysql_timestamp_to_string(MYSQL_TIME *time, char *str)
{
        snprintf(str, DATETIME_LEN, "%4d-%02d-%02d %02d:%02d:%02d", time->year, time-
>month, time->day, time->hour, time->minute, time->second);
}

void mysql_date_to_string(MYSQL_TIME *date, char *str)
{
        snprintf(str, DATE_LEN, "%4d-%02d-%02d", date->year, date->month, date->day);
}
```

dotenv.h

```
#ifndef DOTENV_DOTENV_H
#define DOTENV_DOTENV_H

#include <stdbool.h>

#ifdef __cplusplus
extern "C"
{
#endif

/**
 * @param path Can be a directory containing a file named .env, or the path of the env file itself
 * @param overwrite Existing variables will be overwritten
 * @return 0 on success, -1 if can't open the file
 */
int env_load(const char* path, bool overwrite);

#ifdef __cplusplus
}
#endif

#endif
```

dotenv.c

```c
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>
#include <stdbool.h>

/* strtok_r() won't remove the whole ${ part, only the $ */
#define remove_bracket(name) name + 1

#define remove_space(value) value + 1

static char *concat(char *buffer, char *string)
{
	if (!buffer) {
		return strdup(string);
	}
	if (string) {
		size_t length = strlen(buffer) + strlen(string) + 1;
		char *new = realloc(buffer, length);

		return strcat(new, string);
	}

	return buffer;
}

static bool is_nested(char *value)
{
	return strstr(value, "${") && strstr(value, "}");
}

static char *prepare_value(char *value)
{
	char *new = malloc(strlen(value) + 2);
	sprintf(new, " %s", value);

	return new;
}

static char *parse_value(char *value)
{
	value = prepare_value(value);

	char *search = value, *parsed = NULL, *tok_ptr;
```

```
	char *name;

	if (value && is_nested(value)) {
		while (1) {
			parsed = concat(parsed, strtok_r(search, "${", &tok_ptr));
			name = strtok_r(NULL, "}", &tok_ptr);

			if (!name) {
				break;
			}
			parsed = concat(parsed, getenv(remove_bracket(name)));
			search = NULL;
		}
		free(value);
		return parsed;
	}
	return value;
}

static bool is_commented(char *line)
{
	if ('#' == line[0]) {
		return true;
	}

	int i = 0;
	while (' ' == line[i]) {
		if ('#' == line[++i]) {
			return true;
		}
	}
	return false;
}

static void set_variable(char *name, char *original, bool overwrite)
{
	char *parsed;

	if (original) {
		parsed = parse_value(original);
		setenv(name, remove_space(parsed), overwrite);
		free(parsed);
	}
}
```

```c
static void parse(FILE *file, bool overwrite)
{
	char *name, *original, *line = NULL, *tok_ptr;
	size_t len = 0;

	while (-1 != getline(&line, &len, file)) {
		if (!is_commented(line)) {
			name = strtok_r(line, "=", &tok_ptr);
			original = strtok_r(NULL, "\n", &tok_ptr);

			set_variable(name, original, overwrite);
		}
	}
	free(line);
}

static FILE *open_default(const char *base_path)
{
	char path[strlen(base_path) + strlen(".env") + 1];
	sprintf(path, "%s/.env", base_path);

	return fopen(path, "rb");
}

int env_load(const char *path, bool overwrite)
{
	FILE *file = open_default(path);

	if (!file) {
		file = fopen(path, "rb");

		if (!file) {
			return -1;
		}
	}
	parse(file, overwrite);
	fclose(file);

	return 0;
}
```

io.h

```
#pragma once
#include <stdbool.h>
#include <setjmp.h>

extern jmp_buf leave_buff;
extern bool io_initialized;


#define initialize_io()                 \
__extension__ ({                        \
        io_initialized = true;          \
        int __ret = setjmp(leave_buff);    \
        __ret == 0;                     \
})

extern char *get_input(char *question, int len, char *buff, bool hide);
extern char *get_choice(unsigned int len, char *choice);
extern bool yes_or_no(char *question, char yes, char no, bool default_answer, bool insensitive);
extern int multi_choice(char *question, char *options[], int num);
extern void clear_screen(void);
extern void press_anykey(void);
```

io.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdbool.h>
#include <setjmp.h>
#include <termios.h>

#include "io.h"

jmp_buf leave_buff;
bool io_initialized;

static void leave(void)
{
	if(io_initialized)
		longjmp(leave_buff, 1);
	else
		exit(EXIT_SUCCESS);
}


char *get_input(char *question, int len, char *buff, bool hide)
{
	printf("%s", question);


	struct termios term, oterm;

	if(hide) {
		fflush(stdout);
		if (tcgetattr(fileno(stdin), &oterm) == 0) {
			memcpy(&term, &oterm, sizeof(struct termios));
			term.c_lflag &= ~(ECHO|ECHONL);
			tcsetattr(fileno(stdin), TCSAFLUSH, &term);
		} else {
			memset(&term, 0, sizeof(struct termios));
			memset(&oterm, 0, sizeof(struct termios));
		}
	}

	if(fgets(buff, len, stdin) != NULL) {
		buff[strcspn(buff, "\n")] = 0;
```

```
        } else {
                printf("EOF received, leaving...\n");
                fflush(stdout);
                leave();
        }

        // Empty stdin
        if(strlen(buff) + 1 == len) {
                int ch;
                while(((ch = getchar()) != EOF) && (ch != '\n'));
                if(ch == EOF) {
                        printf("EOF received, leaving...\n");
                        fflush(stdout);
                        leave();
                }
        }
    if(hide) {
                fwrite("\n", 1, 1, stdout);
                tcsetattr(fileno(stdin), TCSAFLUSH, &oterm);
        }

        return buff;
}


 char *get_choice(unsigned int len, char *choice)
{
    char c;
    unsigned int i;

    // Acquisisce da tastiera al più len - 1 caratteri
    for(i = 0; i < len; i++) {
        (void) fread(&c, sizeof(char), 1, stdin);
        if(c == '\n') {
            choice[i] = '\0';
            break;
        } else
            choice[i] = c;
    }

    // Controlla che il terminatore di stringa sia stato inserito
    if(i == len - 1)
        choice[i] = '\0';
```

```
    // Se sono stati digitati più caratteri, svuota il buffer della tastiera
    if(strlen(choice) >= len) {
        // Svuota il buffer della tastiera
        do {
            c = (char)getchar();
        } while (c != '\n');
    }

    return choice;
}


bool yes_or_no(char *question, char yes, char no, bool default_answer, bool insensitive)
{
        int extra;

        // yes and no characters should be lowercase by default
        yes = (char)tolower(yes);
        no = (char)tolower(no);

        // Which of the two is the default?
        char s, n;
        if(default_answer) {
                s = (char)toupper(yes);
                n = no;
        } else {
                s = yes;
                n = (char)toupper(no);
        }

        while(true) {
                printf("%s [%c/%c]: ", question, s, n);
                extra = 0;

                char c = (char)getchar();
                char ch = 0;
                if(c != '\n') {
                        while(((ch = (char)getchar()) != EOF) && (ch != '\n'))
                                extra++;
                }
                if(c == EOF || ch == EOF) {
                        printf("EOF received, leaving...\n");
                        fflush(stdout);
                        leave();
```

```
                }
                if(extra > 0)
                        continue;

                // Check the answer
                if(c == '\n') {
                        return default_answer;
                } else if(c == yes) {
                        return true;
                } else if(c == no) {
                        return false;
                } else if(c == toupper(yes)) {
                        if(default_answer || insensitive) return true;
                } else if(c == toupper(no)) {
                        if(!default_answer || insensitive) return false;
                }
        }
}


int multi_choice(char *question, char *options[], int num)
{
   int i = 0, j = 0, len = 0;

   while(i<num)
      len += (int)strlen(options[i++]);

   char *possibilities = (char *) malloc(len * num * sizeof(char *));

   for(i = 0; i < num; i++) {
      for(int z=0; z<(int)strlen(options[i]); z++) {
         possibilities[j++] = options[i][z];
      }
      possibilities[j++] = '/';
   }
   possibilities[j-1] = '\0';

   while(true) {
      printf("%s [%s]: ", question, possibilities);

      char c[5];
      sprintf(c, "%s", get_choice(5, c));

      if(strlen(c) > 2)
```

```
            continue;

        for (i = 0; i < num; i++) {
            if (strcmp(c, options[i]) == 0) {
                return i;
            }
        }
    }
}


void clear_screen(void)
{
        printf("\033[2J\033[H");
}


void press_anykey(void)
{
        char c;
        puts("\nPremi enter per continuare...");
        while((c = (char)getchar()) != '\n');
        (void)c;
}
```

validation.h

```
#pragma once

#include <stdbool.h>

extern bool init_validation(void);
extern void fini_validation(void);
extern bool validate_time(char *str);
extern bool validate_date(char *str);
```

validation.c

```
#include <stdbool.h>
#include <regex.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

#include "validation.h"

regex_t regex_date;
regex_t regex_time;

static void print_regerror(int errcode, size_t length, regex_t *compiled)
{
	char buffer[length];
	(void) regerror(errcode, compiled, buffer, length);
	fprintf(stderr, "Regex match failed: %s\n", buffer);
}

bool init_validation(void)
{
	int ret1;
	int ret2;

	ret1 = regcomp(&regex_date, "^([12][0-9]{3}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01]))",
REG_EXTENDED);
	if(ret1) {
		if(ret1 == REG_ESPACE) {
			fprintf(stderr, "%s\n", strerror(ENOMEM));
		} else {
			fprintf(stderr, "Fatal error while setting up date validation regex.\n");
		}
	}
```

```
        ret2 = regcomp(&regex_time, "^([0-1]?[0-9]|2?[0-3]|[0-9])$", REG_EXTENDED);
        if(ret2) {
                if(ret2 == REG_ESPACE) {
                        fprintf(stderr, "%s\n", strerror(ENOMEM));
                } else {
                        fprintf(stderr, "Fatal error while setting up time validation regex.\n");
                }
        }

        return ret1 == 0 && ret2 == 0;
}


void fini_validation(void)
{
        regfree(&regex_time);
        regfree(&regex_date);
}

bool validate_date(char *str)
{
        int ret = regexec(&regex_date, str, 0, NULL, REG_NOTEOL);

        if(ret != 0 && ret != REG_NOMATCH) {
                size_t length = regerror(ret, &regex_date, NULL, 0);
                print_regerror(ret, length, &regex_date);
                return false;
        }
        return ret == 0;
}

bool validate_time(char *str)
{
        int ret = regexec(&regex_time, str, 0, NULL, 0);

        if(ret != 0 && ret != REG_NOMATCH) {
                size_t length = regerror(ret, &regex_time, NULL, 0);
                print_regerror(ret, length, &regex_time);
                return false;
        }
        return ret == 0;
}
```