PROGETTO PALESTRA

Gruppo **Rollbackers**: Cailotto Massimo, matricola 880763
Minardi Matteo, matricola 880895

Indice:

Introduzione

Il progetto si basa sulla creazione di un'applicazione web per la gestione ed il controllo degli accessi tramite prenotazione online di una palestra.
La web application si interfaccia con un database SQL sottostante progettato ad hoc proprio per questo scopo ed integrato utilizzando librerie di Flask e SQLAlchemy in linguaggio Python.
Attraverso un calendario suddiviso in slot prenotabili, i clienti della palestra possono prenotare le proprie sessioni di allenamento, che possono essere individuali in una sala pesi oppure di gruppo in una sala corsi.
Ogni cliente ha a propria disposizione un'area riservata dove saranno riassunte le sue informazioni personali e dove potrà prenotare gli slot precedentemente citati; un cliente inoltre potrà visualizzare lo storico delle proprie prenotazioni effettuate, un abbonato potrà scegliere se cancellarne qualcuna futura, ed uno non abbonato potrà effettuare un abbonamento.
L'amministratore sarà in grado di modificare, secondo le proprie preferenze o gli obblighi da parte del Ministero della Salute, il numero di accessi settimanali ed il numero di slot giornalieri prenotabili per ogni abbonato, il numero di persone massime che possono presentarsi all'interno della stessa fascia oraria e il numero di metri quadri di stanza minimi che si vogliono dedicare ad ogni cliente presente.
L'organizzazione della struttura della palestra (stanze, corsi e sale pesi) ed i suoi membri dello staff è gestita dall'amministratore attraverso una pagina di controllo dalla quale potrà inserire tutte le personalizzazioni che desidera o che necessita, dall'aggiunta alla modifica, alla rimozione.

Funzionalità principali

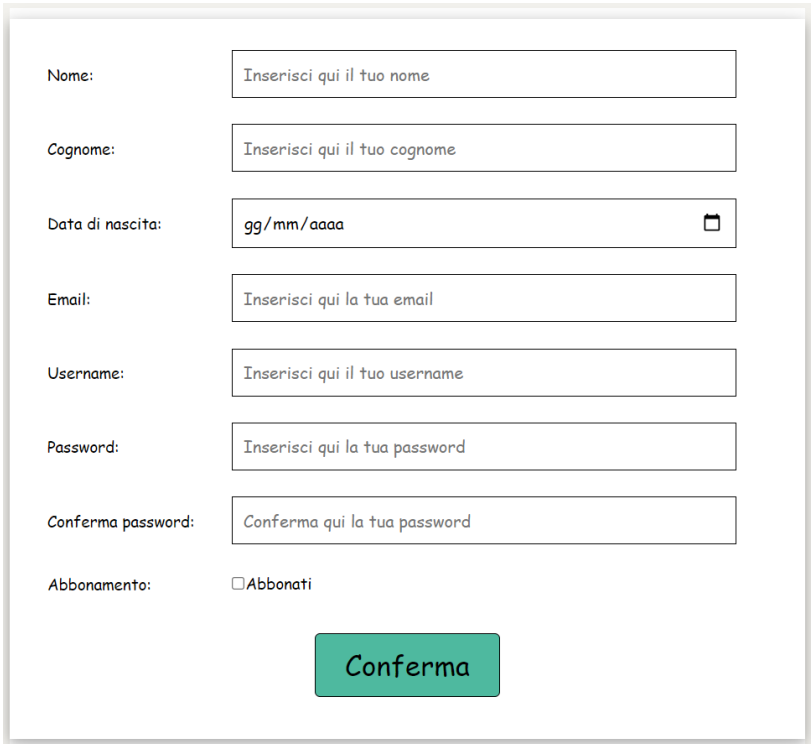
Visualizzazioni informazioni palestra

Mostra in forma tabellare la composizione della palestra

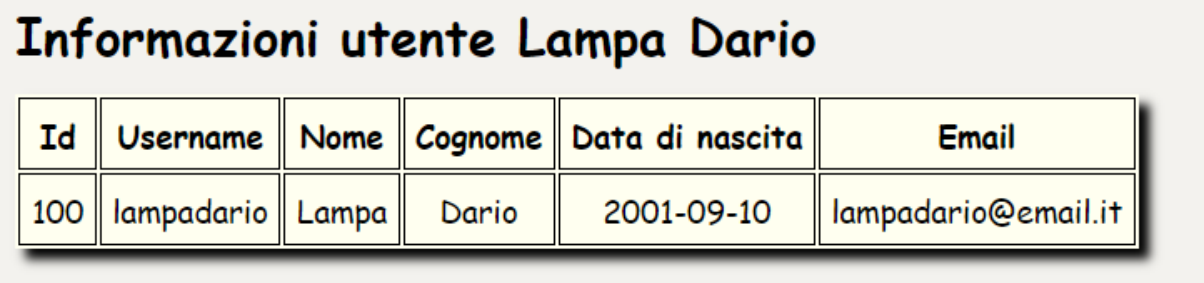
Per gli utenti:

Creazione di un account

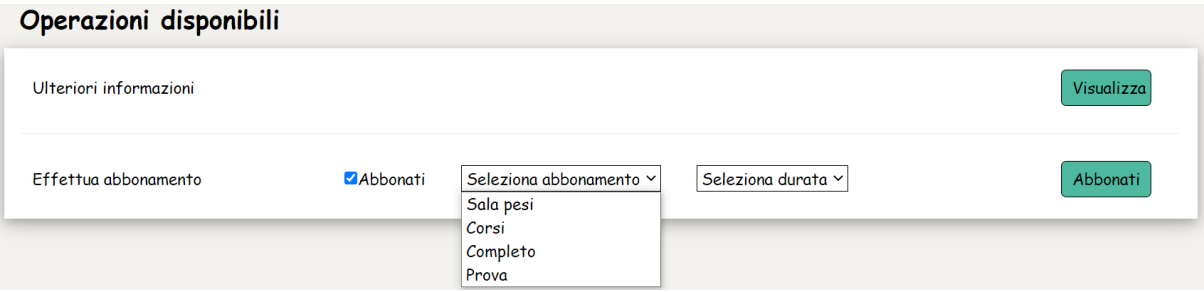
Con la possibilità di inserire le proprie informazioni personali e di contatto, e di abbonarsi immediatamente



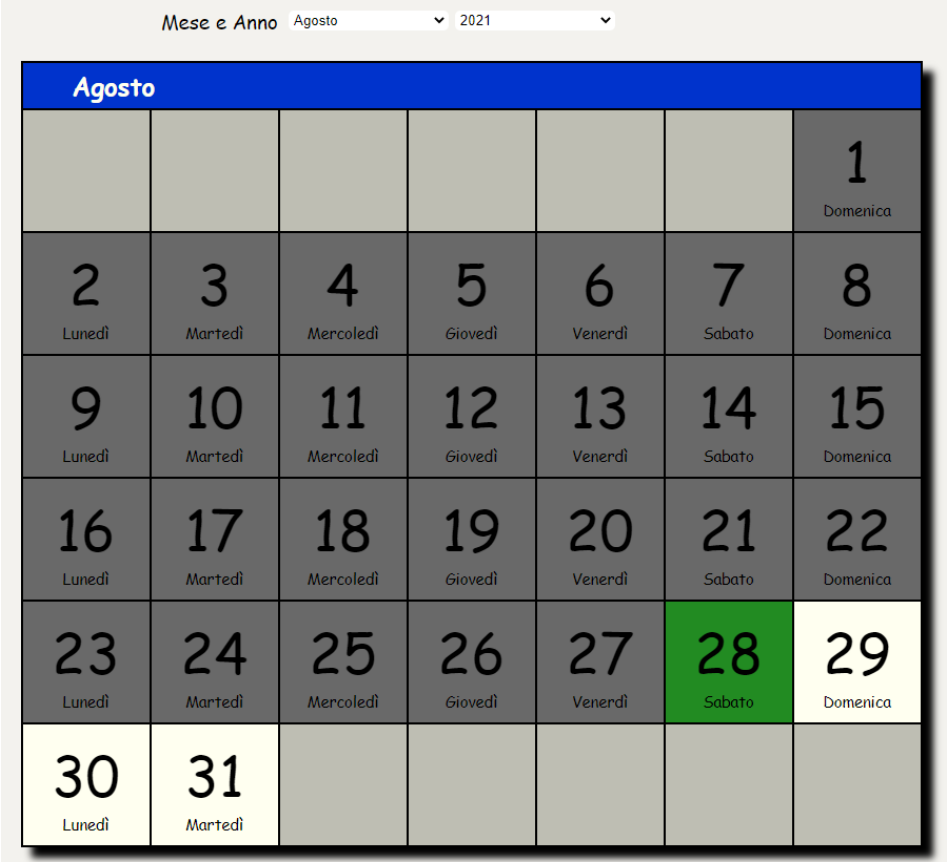
Visualizzare le proprie informazioni personali e di contatto
Per avere la conferma che siano corrette



Abbonarsi, se non già fatto al momento della registrazione dell'account
Possibilità di specificare la durata dell'abbonamento tra alcune proposte (1 mese, 3 mesi, 6 mesi ed 1 anno).
C'è anche la possibilità di avere un abbonamento speciale di prova dalla durata di una settimana.

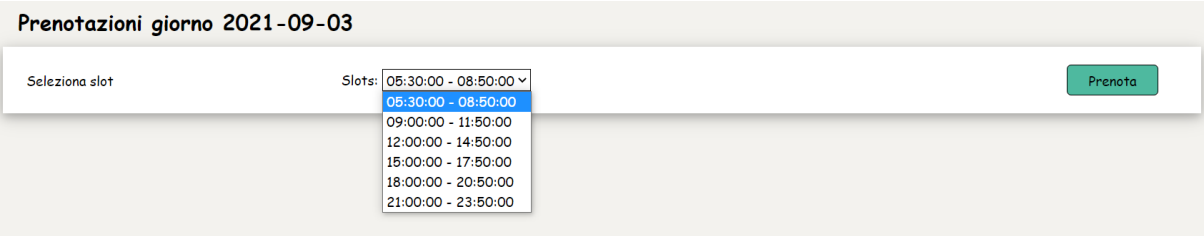


Prenotazione di un numero massimo prefissato di slot al giorno, una quantità finita di volte a settimana
Con l'aggiunta di opportuni controlli, è possibile prenotare solo slot nel futuro e, per motivi organizzativi della palestra, unicamente appartenenti ai prossimi sette giorni.
La data corrente è indicata in verde, i giorni non disponibili sono colorati di grigio, quelli disponibili di bianco.
La prenotazione potrà proseguire solamente se l'utente non ha effettuato troppi abbonamenti in quella specifica data o in quella intera settimana.



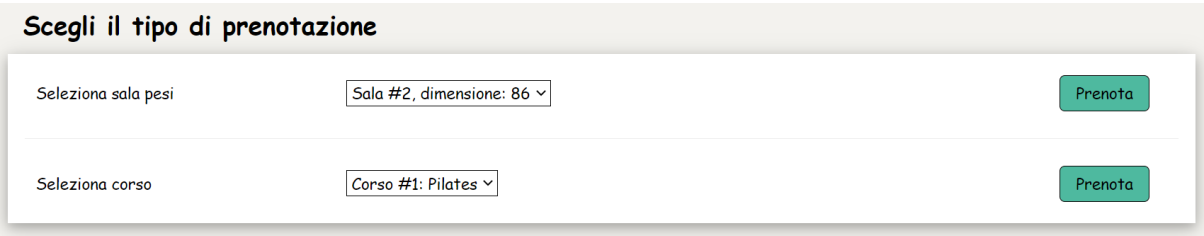
Per ogni giorno si ha la possibilità di scegliere l'orario di allenamento tra sei opzioni predefinite

La prenotazione potrà proseguire solo se l'utente non ha già effettuato una prenotazione per lo slot specifico selezionato

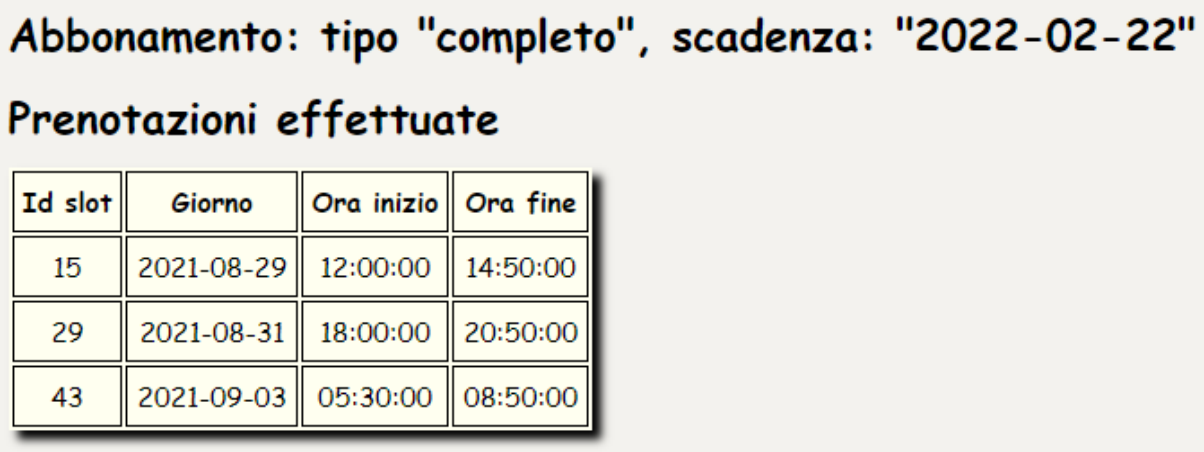


Per ogni slot si può scegliere di partecipare unicamente alla sala pesi o ai corsi disponibili in quel determinato orario

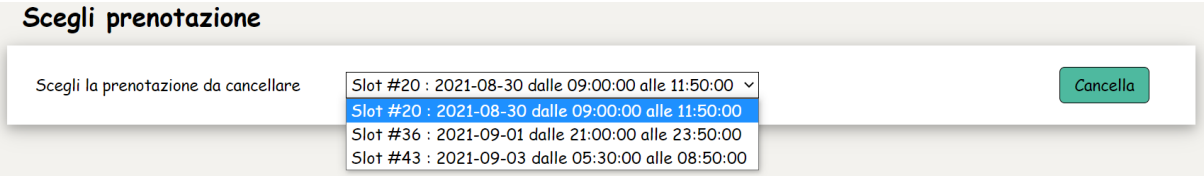
La prenotazione potrà proseguire solo se il numero di persone già precedentemente prenotate per l'attività selezionata è strettamente minore del numero massimo di persone che possono parteciparvi.



Visualizzazione del proprio storico di prenotazioni in ordine di data e orario



Possibilità di cancellare quelle future alle quali non si ha più intenzione o possibilità di partecipare.



Per l'amministratore:

Effettuare check periodici (controlli giornalieri) che garantiscano automaticamente la corretta struttura della base di dati

- controllo che esistano i giorni in cui saranno effettuate le attività, con conseguente creazione in caso di esito negativo
- controllo che esistano gli slot per ogni orario della giornata, con conseguente creazione in caso di esito negativo
- aggiornamento delle tabelle abbonati e nonabbonati in base alle scadenze degli abbonamenti degli utenti
- aggiornamento degli iscritti massimi delle attività in modo che rispettino la dimensione della stanza in cui si svolgono
- controllo che il numero di iscritti possibili dato dalla somma di tutte le attività in uno slot non superi il numero di persone massime di quello slot
- controllo che i corsi e le sale pesi siano sempre in slot prenotabili

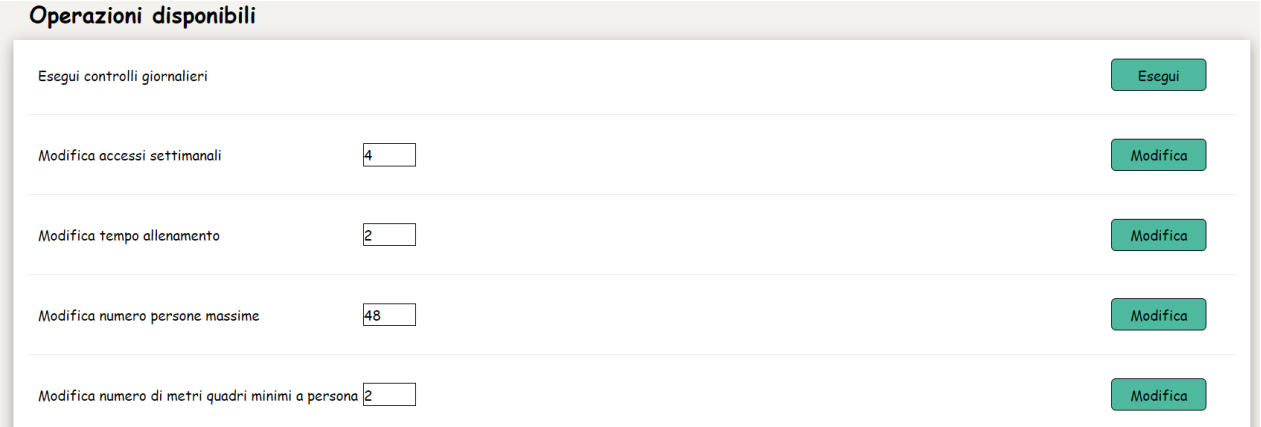
- **Decidere un numero massimo di accessi settimanali in giorni distinti** che un abbonato può prenotare dal calendario

- **Decidere un tempo massimo di allenamento giornaliero** che un abbonato può prenotare dal calendario

- **Personalizzare il numero di persone massime che possono accedere alla palestra in uno slot**

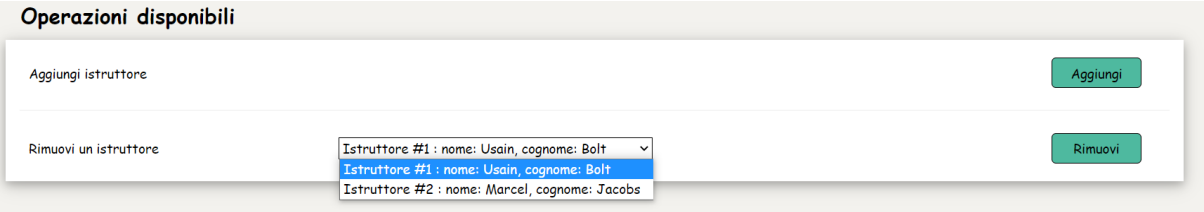
Questo valore può solo essere abbassato in quanto fare altrimenti porterebbe ad una violazione del vincolo seguente, infatti si consiglia di eseguirlo solamente dopo aver scelto un numero minimo di metri quadri a persona..

- **Personalizzare il numero di metri quadri minimi da garantire ad ogni persona**



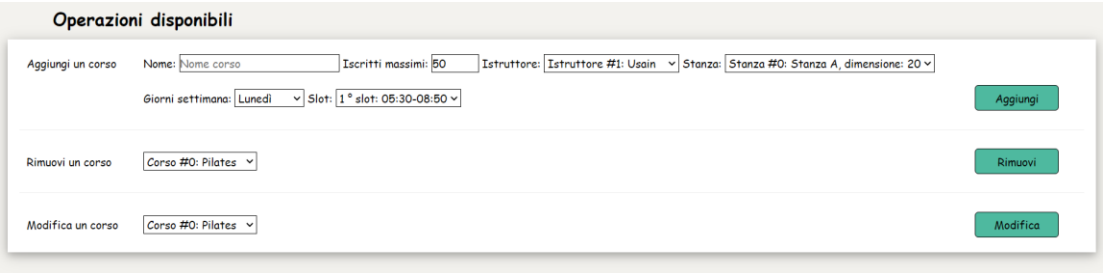
Gestione personale

Possibilità di specificare la stessa quantità di informazioni dei clienti
Gli istruttori devono essere aggiunti prima dei corsi che intendono insegnare



Gestione corsi

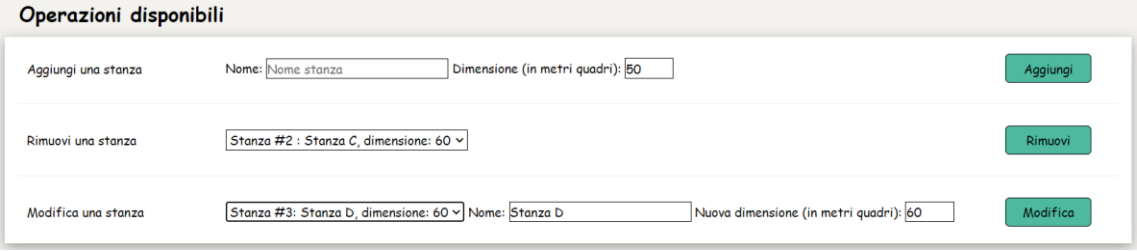
Alla creazione c'è la possibilità di personalizzare le informazioni principali: nome, istruttore, stanza, giorno e orario settimanale.
Il database si occuperà automaticamente di aggiungere le sedute prenotabili all'interno del calendario.
Cancellare un corso comporta l'eliminazione di tutte le sue sedute e delle relative prenotazioni future effettuate dai clienti.



Gestione stanze

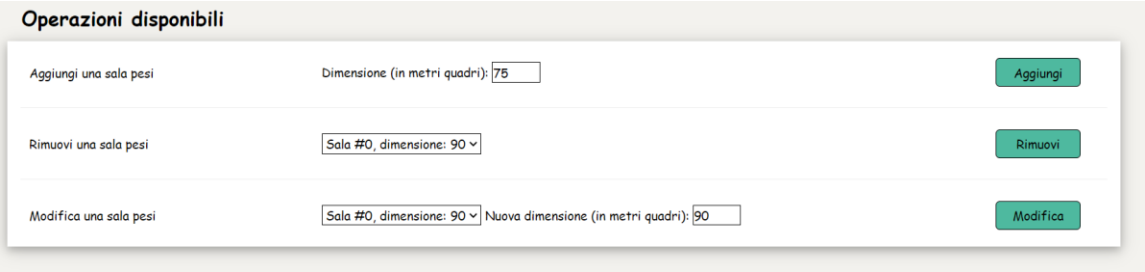
Possibile specificare nome e dimensione sia in creazione che in modifica
Grazie ad un trigger, la rimozione di una stanza nella quale si svolge un corso, non è

possibile e non ha alcun effetto.



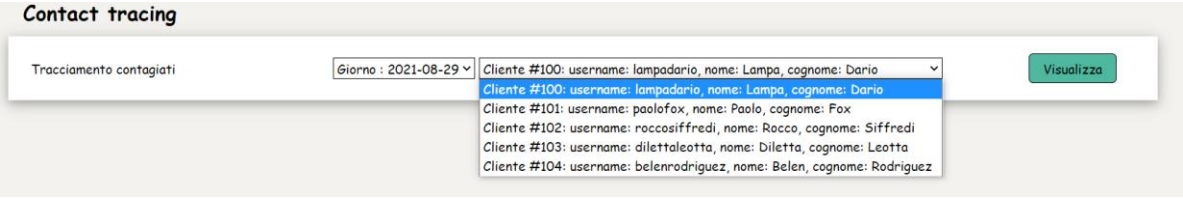
Gestione salepesi

Possibile specificare dimensione sia in creazione che in modifica.
Il database si occuperà automaticamente di aggiungere le sedute prenotabili all'interno del calendario.
Cancellare una sala pesi comporta l'eliminazione di tutte le sue sedute e delle relative prenotazioni future effettuate dai clienti.



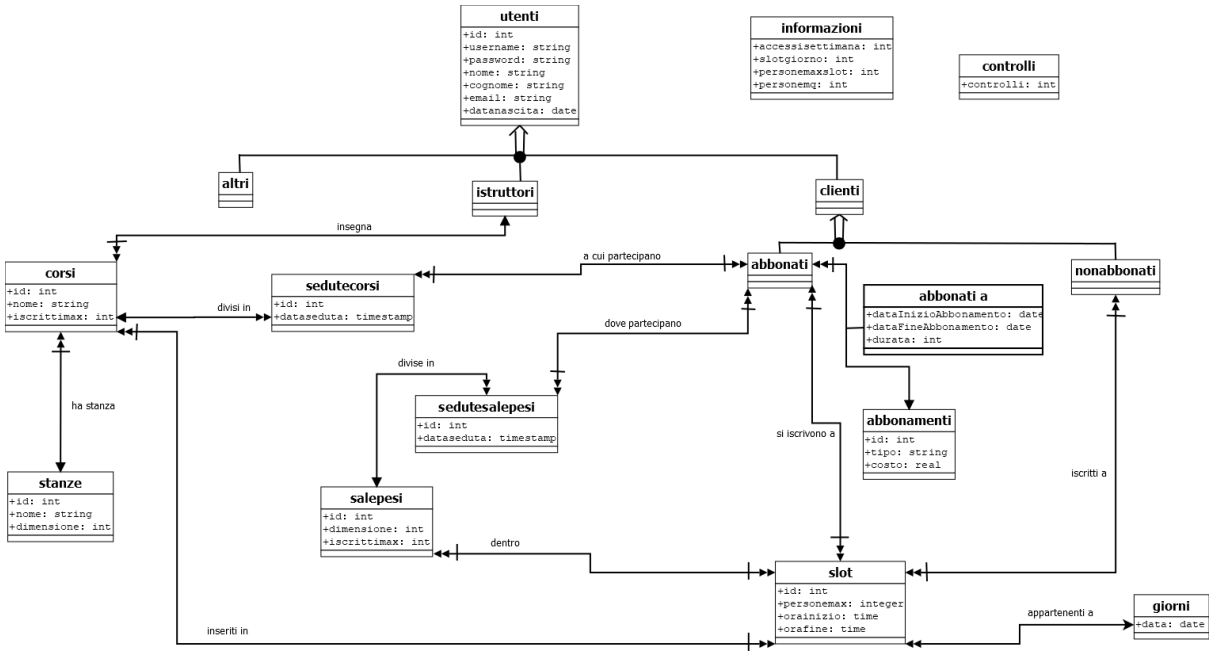
Possibilità di visualizzare i clienti entrati in contatto con contagiati covid

Scelta di un giorno da usare come inizio del possibile contagio e un utente come paziente zero.
Verranno visualizzati tutti i clienti (sia abbonati che non abbonati) che hanno effettuato almeno una prenotazione in comune in uno slot con il paziente zero, dalla data selezionata a quella corrente.

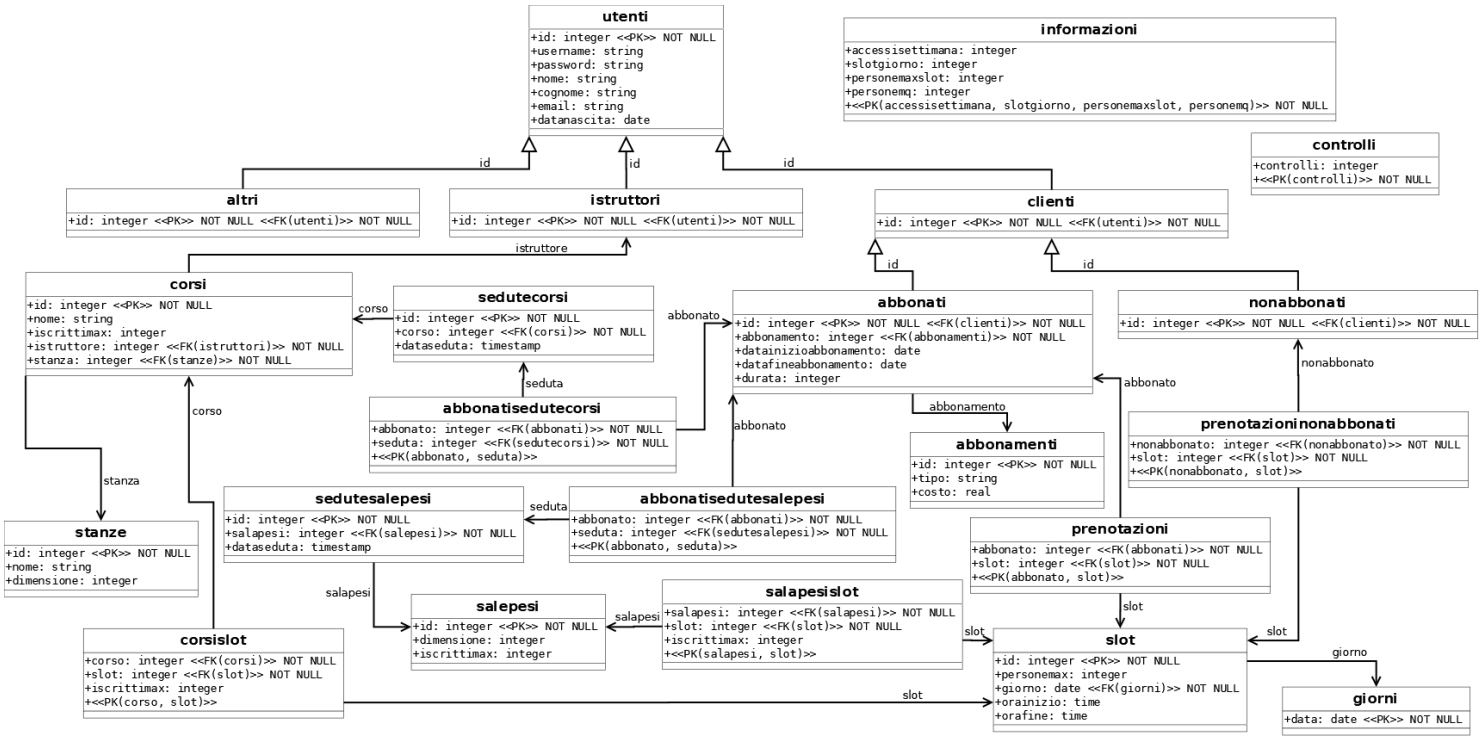


Progettazione concettuale e logica

Schema concettuale



Schema logico



Nello schema logico (relazionale) sono presenti tutte le classi con i relativi attributi scelti opportunamente.

Le classi **informazioni** e **controlli** sono classi scollegate rispetto alle altre poichè di carattere generale rispetto all'applicazione. Inoltre non sarebbe corretto collegarle al resto delle classi vista la loro indipendenza dal contesto.

La classe **informazioni** è una classe contenente le informazioni generali più importanti come gli accessi settimanali disponibili per ogni cliente, il numero massimo di slot prenotabili al giorno, il numero di metri quadri di spazio garantiti ad ogni persona.
La classe **controlli** contiene un unico attributo che appena viene modificato attiva un trigger specifico utile ad effettuare i controlli periodici per mantenere il database aggiornato.
All'interno dello schema sono presenti 2 gerarchie di classi:

- La classe **utenti** rappresenta generalmente le persone che compongono il nostro universo di interesse e ne specifica alcuni dati personali ed informazioni di contatto. Essa ha 3 sottoclassi (**altri**, **istruttori** e **clienti**) e viene utilizzato il partizionamento verticale senza l'aggiunta di ulteriori attributi, dato che tutte le informazioni utili sono già presenti direttamente all'interno della classe **utenti.**
- La classe **clienti** rappresenta i clienti effettivi registrati all'interno dell'applicazione. Ha 2 sottoclassi (**abbonati** e **nonabbonati**), in modo da non perdere le informazioni dei clienti che non sono ancora abbonati o che non lo sono più, e quindi non dovranno ricreare un proprio account in futuro.
  Viene utilizzato il partizionamento verticale con l'aggiunta di attributi utili per le informazioni riguardanti l'abbonamento, rappresentato dalla classe **abbonamenti**, nella classe **abbonati.**

La classe **corsi** contiene tutte le informazioni relative ai vari corsi presenti all'interno della palestra, oltre ad un nome ed ad un istruttore, viene specificato anche il numero di iscritti massimi che quel corso può avere in base al numero di metri quadri minimi da garantire ad ogni persona e alla dimensione dello spazio in cui il corso si svolge; è quindi collegata tramite un'associazione alla classe **stanze** che rappresenta proprio quel luogo nella palestra.
La classe **salepesi** rappresenta le sale pesi presenti nella palestra, ha anch'essa l'attributo iscrittimax che rappresenta il numero massimo di iscritti in base alla sua dimensione e, ugualmente ai corsi, al numero di metri quadri minimi da dedicare ad ogni abbonato.
Ad ogni corso sono associate più sedute, rappresentate dalla classe **sedutecorsi** ed ogni cliente ha la possibilità di prenotarsi a ciascuna di esse memorizzando tutto all'interno della classe **abbonatisedutecorsi**.
Un funzionamento analogo è quello classi **sedutesalepesi** e **abbonatisedutesalepesi**.
La classe giorni contiene le date rilevanti che vale la pena tenere memorizzate all'interno della base di dati.
Ogni giorno è suddiviso in esattamente sei slot prenotabili dai clienti.
Le informazioni relative alle prenotazioni da parte degli abbonati vengono memorizzate nella classe **prenotazioni**, mentre per non perdere quelle effettuate da utenti a cui è scaduto l'abbonamento negli ultimi sette giorni, e dunque mantenere uno storico, salviamo in **prenotazioninonabbonati** le loro prenotazioni.
Gli slot, rappresentati dalla classe **slot**, contengono oltre all'ora di inizio e di fine l'attributo personemax che definisce il numero massimo di persone che si possono prenotare per quel determinato slot.
Le tabelle **corsislot** e **salapesislot** servono per associare ad ogni coppia corso-slot o salapesi-slot il loro corrispondente numero di iscritti massimi, che può variare in base al numero di attività presenti in quello slot.

Query principali

Funzione create_admin

```
def create_admin():
    exists = session.query(classes.Other).filter(classes.Other.id == 0).first()
    if not exists:
        new_id = 0
        pw = 'admin' + 'admin@palestra.it'
        user = classes.User(id=new_id, username="admin", password=hashlib.md5(pw.encode()).hexdigest(),
                            nome="admin", cognome="admin", email="admin@palestra.it",
                            datanascita='1000-01-01')
        admin = classes.Other(new_id)
        session.add(user)
        session.add(admin)
        session.commit()
```

Versione SQL

```
DECLARE exist altri%rowtype;
SELECT * INTO exist FROM altri WHERE id=0;
IF NOT exist THEN
    INSERT INTO utenti VALUES(0, "admin",
hashlib.md5(pw.encode()).hexdigest(), "admin", "admin",
"admin@palestra.it", '1000-01-01');
    INSERT INTO altri VALUES(0);
END IF;
```

La funzione/query crea (solo se non esiste) l'utente "admin" che gestirà quindi la parte di amministrazione legata all'applicazione, con la possibilità di aggiungere, modificare e rimuovere informazioni di carattere generale e classi specifiche.

Getters

Funzione get_user_by_email

```
def get_user_by_email(email):
    user = session.query(classes.User).filter(classes.User.email == email).first()
    return classes.User(user.id, user.username, user.password, user.nome, user.cognome, user.email, user.datanascita)
```

Versione SQL

```
SELECT * FROM utenti WHERE email = email
```

La funzione/query restituisce l'utente in basa alla mail passata come parametro.

Funzione get_subscriber_by_id

```
def get_subscriber_by_id(id):
    subscriber = session.query(classes.Subscriber).filter(classes.Subscriber.id == id).first()
    if subscriber:
        return classes.Subscriber(subscriber.id, subscriber.abbonamento, subscriber.datafineabbonamento,
                                  subscriber.datafineabbonamento, subscriber.durata)
    else:
        return None
```

Versione SQL

```
SELECT * FROM abbonati WHERE id = id
```

La funzione/query ritorna l'abbonato in base all'id passato.

Funziona allo stesso modo:

- **get_subscription_by_id**
- **get_course**

Funzione get_id_increment

```
def get_id_increment():
    user = session.query(classes.User).filter(classes.User.id >= 100).order_by(classes.User.id.desc()).first()
    if user:
        return user.id + 1
    else:
        return first_id_client
```

Versione SQL

```
SELECT * FROM utenti WHERE id>=100 ORDER BY id DESC
```

La funzione/query ritorna l'id corretto per la creazione di un nuovo cliente, incrementando di 1 quindi l'ultimo id utile.
Per scelte implementative gli id dei clienti partono dal numero 100.

Funzione get_id_staff_increment

```
def get_id_staff_increment():
    user = session.query(classes.User).filter(classes.User.id < 100).order_by(classes.User.id.desc()).first()
    if user:
        return user.id + 1
    else:
        return first_id_client
```

Versione SQL

```
SELECT * FROM utenti WHERE id<100 ORDER BY id DESC
```

La funzione/query rispetto a quella precedente ritorna l'id corretto per la creazione di un nuovo membro dello staff prelevando in caso l'id dagli id minori di 100.

Funzionano allo stesso modo:

- **get_course_id_increment**
- **get_room_id_increment**
- **get_weight_room_id_increment**

Funzione get_courses

```
def get_courses():
    courses = session.query(classes.Course).order_by(classes.Course.id.asc()).all()
    return courses
```

Versione SQL

```
SELECT * FROM corsi ORDER BY id ASC
```

La funzione ritorna tutti i corsi, in questo caso ordinati per id.

Funzionano allo stesso modo:

- **get_rooms**
- **get_weight_rooms**
- **get_trainers**
- **get_others**
- **get_clients**
- **get_information**
- **get_checks**

Funzione get_slot_from_date

```
def get_slot_from_date(data):
    slots = session.query(classes.Slot).filter(classes.Slot.giorno == data).order_by(classes.Slot.orainizio).all()
    return slots
```

Versione SQL

```
SELECT * FROM slot WHERE giorno = DATE data ORDER BY orainizio
```

La funzione/query restituisce tutti gli slot in cui è suddiviso il giorno passato come parametro.

Funzione get_slot_weight_rooms

```
def get_slot_weight_rooms(idSlot, subscription):
    if subscription == 'corsi':
        return []
    else:
        weightrooms = session.query(classes.WeightRoom).filter(classes.WeightRoom.id.in_(
            session.query(classes.WeightRoomSlot.salapesi).filter(classes.WeightRoomSlot.slot == idSlot)
        )).all()
        return weightrooms
```

Versione SQL

```
SELECT * FROM salepesi WHERE id IN (
    SELECT sp.salapesi FROM salapesislot sp WHERE sp.slot=idSlot)
```

La funzione/query ritorna tutte le sale pesi che sono prenotabili in un certo slot passato come parametro.

Funziona allo stesso modo:

- **get_slot_courses**

Funzione get_coursesitting_id

```
def get_coursesitting_id(idSlot, idCorso):
    giorno = session.query(classes.Slot.giorno).filter(classes.Slot.id == idSlot)
    id_sitting = session.query(classes.CourseSitting.id).\
        filter(and_(func.DATE(classes.CourseSitting.dataseduta) == giorno, classes.CourseSitting.corso == idCorso)).\
        first()
    return id_sitting
```

Versione SQL

```
SELECT id FROM sedutecorsi WHERE (dataseduta::date)=(
    SELECT giorno FROM slot WHERE id = idSlot)
    AND corso = idCorso
```

La funzione/query restituisce l'id della seduta effettuata del corso avente id passato come parametro, in base allo slot del giorno prenotato passato come parametro.

Funziona allo stesso modo:

- **get_weight_room_sitting_id**

Funzione get_resevations

```
def get_reservations(idSub):
    reservations = session.query(classes.Reservation.slot, classes.Slot.giorno, classes.Slot.orainizio,
                                 classes.Slot.orafine).\
        filter(and_(classes.Reservation.slot == classes.Slot.id, classes.Reservation.abbonato == idSub,
                    classes.Slot.giorno > func.current_date())).\
        order_by(classes.Slot.giorno, classes.Slot.orainizio).all()
    return reservations
```

Versione SQL

```
SELECT p.slot, s.giorno, s.orainizio, s.orafine
FROM prenotazioni p JOIN slot s ON p.slot=s.id
WHERE p.abbonato = idSub AND s.giorno > CURRENT_DATE
ORDER BY s.giorno, s.orainizio
```

La funzione/query ritorna le prenotazioni effettuate dal cliente passato come parametro.

Funzione get_infected

```
def get_infected(giorno, infetto):
    slots1 = session.query(classes.Reservation.slot).\
        filter(and_(classes.Reservation.abbonato == infetto, classes.Reservation.slot == classes.Slot.id,
                    classes.Slot.giorno >= giorno))
    slots2 = session.query(classes.NSReservation.slot).\
        filter(and_(classes.NSReservation.nonabbonato == infetto, classes.NSReservation.slot == classes.Slot.id,
                    classes.Slot.giorno >= giorno))
    slots = slots1.union(slots2)
    sub_infected = session.query(classes.User).\
        filter(and_(classes.User.id != infetto, classes.User.id == classes.Subscriber.id,
                    classes.User.id == classes.Reservation.abbonato, classes.Reservation.slot.in_(slots)))
    notsub_infected = session.query(classes.User).\
        filter(and_(classes.User.id != infetto, classes.User.id == classes.NotSubscriber.id,
                    classes.User.id == classes.NSReservation.nonabbonato, classes.NSReservation.slot.in_(slots)))
    infected = sub_infected.union(notsub_infected).all()
    return infected
```

Versione SQL

```
(SELECT * FROM utenti u JOIN abbonati a ON u.id=a.id JOIN prenotazioni
p ON p.abbonato=u.id WHERE u.id <> infetto AND p.slot IN (
SELECT p.slot FROM prenotazioni p JOIN slot s ON p.slot=s.id WHERE
p.abbonato = infetto AND s.giorno >= giorno
UNION
SELECT p.slot FROM prenotazioninonabbonati p JOIN slot s ON p.slot=s.id
WHERE p.abbonato = infetto AND s.giorno >= giorno))
UNION
(SELECT * FROM utenti u JOIN nonabbonati a ON u.id=a.id JOIN
prenotazioninonabbonati p ON p.nonabbonato=u.id WHERE u.id <> infetto
AND p.slot IN (
SELECT p.slot FROM prenotazioni p JOIN slot s ON p.slot=s.id WHERE
p.abbonato = infetto AND s.giorno >= giorno
UNION
SELECT p.slot FROM prenotazioninonabbonati p JOIN slot s ON p.slot=s.id
WHERE p.abbonato = infetto AND s.giorno >= giorno))
```

La funzione/query restituisce le persone che potrebbero essere state contagiate in un determinato giorno dal cliente passato come parametro.

Funzione is_reserved

```
def is_reserved(user_id, idSlot):
    res = session.query(classes.Reservation).\
        filter(and_(classes.Reservation.abbonato == user_id, classes.Reservation.slot == idSlot)).first()
    if res:
        return True
    else:
        return False
```

Versione SQL

```
SELECT * FROM prenotazioni WHERE abbonato = user_id AND slot = idSlot
```

La funzione/query controlla se è già presente una prenotazione del cliente passato come parametro in un certo slot.

Funzione is_available_slot

```
def is_available_slot(idSlot):
    res = session.query(func.count(classes.Reservation.abbonato)).filter(classes.Reservation.slot == idSlot).first()
    personemax = session.query(classes.Slot.personemax).filter(classes.Slot.id == idSlot).first()
    if res < personemax:
        return True
    else:
        return False
```

Versione SQL

```
SELECT COUNT(abbonato) FROM prenotazioni WHERE slot = idSlot
SELECT personemax FROM slot WHERE id = idSlot
```

La funzione/query controlla se è possibile prenotarsi ad un determinato slot, controllando se il numero di persone che hanno effettuato già una prenotazione sia minore del numero massimo di persone che si possono prenotare in quello slot.

Funzione is_available_course

```python
def is_available_course(idSeduta, idSlot):
    res = session.query(func.count()).filter(classes.SubscriberCourseSession.seduta == idSeduta).first()
    corso = session.query(classes.CourseSitting.corso).filter(classes.CourseSitting.id == idSeduta)
    iscrittimax_corso = session.query(classes.Course.iscrittimax).filter(classes.Course.id == corso).first()
    iscrittimax_corsislot = session.query(classes.CourseSlot.iscrittimax).\
        filter(and_(classes.CourseSlot.corso == corso, classes.CourseSlot.slot == idSlot)).first()
    if res < iscrittimax_corso and res < iscrittimax_corsislot:
        return True
    else:
        return False
```

Versione SQL

```sql
SELECT COUNT(*) FROM abbonatisedutecorsi WHERE seduta = idSeduta
SELECT iscrittimax FROM corsi WHERE id = (
    SELECT corso FROM sedutecorsi WHERE id = idSeduta)
SELECT iscrittimax FROM corsislot WHERE corso = (
    SELECT corso FROM sedutecorsi WHERE id = idSeduta) AND slot =
idSlot
```

La funzione/query controlla che sia possibile effettuare la prenotazione ad una seduta di un corso, passata come parametro, in un determinato slot.
Perché ciò si verifichi il numero di persone iscritte alla seduta deve essere strettamente minore del numero di persone che quel corso può offrire e del numero di persone che possono accedere a quel corso nello slot passato come parametro.

Funziona allo stesso modo:

- **is_available_weight_room**

Funzione has_exceeded_accessisettimana

```python
def has_exceeded_accessisettimana(user_id, giorno):
    res1 = session.query(func.extract('dow', func.DATE(giorno))).first()
    if int(str(res1)[1:2]) > 0:
        res2 = session.query(func.count(classes.Slot.giorno.distinct())).\
            filter(and_(classes.Reservation.slot == classes.Slot.id, classes.Reservation.abbonato == user_id,
                        classes.Slot.giorno > func.DATE(giorno) - int(str(res1)[1:2]),
                        classes.Slot.giorno <= func.DATE(giorno) + (7 - int(str(res1)[1:2])),
                        classes.Slot.giorno != func.DATE(giorno))).first()
    else:
        res2 = session.query(func.count(classes.Slot.giorno.distinct())).\
            filter(and_(classes.Reservation.slot == classes.Slot.id, classes.Reservation.abbonato == user_id,
                        classes.Slot.giorno >= func.DATE(giorno) - 6, classes.Slot.giorno <= func.DATE(giorno),
                        classes.Slot.giorno != func.DATE(giorno))).first()
    res3 = session.query(classes.Information.accessisettimana).first()
    if res2 < res3:
        return False
    else:
        return True
```

Versione SQL

```
SELECT EXTRACT(DOW FROM DATE giorno)
SELECT COUNT(DISTINCT(s.giorno)) FROM prenotazioni p JOIN slot s ON
p.slot=s.id
WHERE abbonato = user_id AND s.giorno > DATE giorno - %s AND s.giorno
<= DATE giorno + (7 - %s) AND s.giorno <> giorno

SELECT COUNT(DISTINCT(s.giorno)) FROM prenotazioni p JOIN slot s ON
p.slot=s.id
WHERE abbonato = user_id AND s.giorno >= DATE giorno - 6 AND s.giorno
<= DATE giorno AND s.giorno <> giorno
SELECT accessisettimana FROM informazioni
```

La funzione/query controlla che il numero di prenotazioni in giorni differenti effettuate dal cliente passato come parametro sia strettamente minore del numero di accessi settimanali deciso dall'amministrazione.

Funzione has_exceeded_slotgiorno

```
def has_exceeded_slotgiorno(user_id, giorno):
    res1 = session.query(func.count()).\
        filter(and_(classes.Reservation.slot == classes.Slot.id, classes.Reservation.abbonato == user_id,
                    classes.Slot.giorno == func.DATE(giorno))).first()
    res2 = session.query(classes.Information.slotgiorno).first()
    if res1 >= res2:
        return True
    else:
        return False
```

Versione SQL

```
SELECT COUNT(*) FROM prenotazioni p JOIN slot s ON p.slot=s.id WHERE
p.abbonato = user_id AND s.giorno = giorno
SELECT slotgiorno FROM informazioni
```

La funzione/query controlla che il numero di slot prenotati dal cliente passato come parametro in un certo giorno sia strettamente minore del numero di slot prenotabili deciso dall'amministrazione.

Funzione set_checks

```
def set_checks(controlliGiornalieri):
    session.query(classes.Checks).update({"controllo": controlliGiornalieri})
```

Versione SQL

```
UPDATE controlli SET controllo = controlliGiornalieri
```

La funzione/query effettua l'update della tabella controlli aggiornando l'attributo controllo con il valore passato come parametro.

Funzionano allo stesso modo:

- **set_information_accessisettimana**
- **set_information_slotgiorno**
- **set_information_personemaxslot**
- **set_information_personemq**
- **update_weight_room**
- **update_room**
- **update_course**

Funzione remove_room

```
def remove_room(idStanza):
    session.query(classes.Room).filter(classes.Room.id == idStanza).delete()
```

Versione SQL

```
DELETE FROM stanze WHERE id = idStanza
```

La funzione/query rimuove la stanza avente id passato come parametro.

Funzionano allo stesso modo:

- **remove_weight_room**
- **remove_course**
- **remove_user**
- **remove_not_subscriber**

Funzione remove_reservation

```
def remove_reservation(idSub, idSlot):
    session.query(classes.Reservation).\
        filter(and_(classes.Reservation.abbonato == idSub, classes.Reservation.slot == idSlot)).delete()
    if is_reserved_course(idSub, idSlot):
        session.query(classes.SubscriberCourseSession).\
            filter(and_(classes.SubscriberCourseSession.abbonato == idSub,
                        classes.SubscriberCourseSession.seduta == classes.CourseSitting.id,
                        classes.CourseSitting.corso == classes.Course.id, classes.Course.id == classes.CourseSlot.corso,
                        classes.CourseSlot.slot == idSlot)).delete(synchronize_session='fetch')
    else:
        session.query(classes.SubscriberWeightRoomSession).\
            filter(and_(classes.SubscriberWeightRoomSession.abbonato == idSub,
                        classes.SubscriberWeightRoomSession.seduta == classes.WeightRoomSitting.id,
                        classes.WeightRoomSitting.salapesi == classes.WeightRoom.id,
                        classes.WeightRoom.id == classes.WeightRoomSlot.salapesi,
                        classes.WeightRoomSlot.slot == idSlot)).delete(synchronize_session='fetch')
```

Versione SQL

```
DELETE FROM prenotazioni WHERE abbonato = idSub AND slot = idSlot
DELETE FROM abbonatisedutecorsi
WHERE abbonato = idSub AND seduta IN (
                SELECT id FROM sedutecorsi WHERE corso IN (
                    SELECT id FROM corsi WHERE id IN (
                        SELECT corso FROM corsislot WHERE slot =
idSlot)))
DELETE FROM abbonatisedutesalepesi
WHERE abbonato = idSub AND seduta IN (
                SELECT id FROM sedutesalepesi WHERE salapesi IN (
```

```
                    SELECT id FROM salapesi WHERE id IN (
                        SELECT salapesi FROM salapesislot WHERE slot =
idSlot)))
```

La funzione/query elimina la prenotazione effettuata da un certo cliente in un determinato slot.
Controllando il tipo di abbonamento del cliente vengono eliminate anche tutte le righe corrispondenti nelle tabelle relative alle sedute dei corsi e delle sale pesi.

Principali scelte progettuali

Organizzazione della base di dati

Per motivi organizzativi gli id degli utenti registrati all'interno dell'applicazione sono suddivisi in due intervalli:

- Da 0 a 100 riservati per lo staff della palestra
- Maggiori o uguali a 100 riservati per i clienti della palestra

L'id 0 è riservato all'utente di amministrazione.

Nel database sono presenti due tabelle (informazioni e controlli) scollegate da tutte le altre, questo perché esse rappresentano delle informazioni indipendenti dalla struttura fisica della palestra da rappresentare, e si occupano della gestione più astratta dei dati.
Alcune invarianti nella base di dati vengono gestite attraverso trigger (ad esempio il fatto che gli username e le email debbano essere sempre diverse da utente ad utente) mentre altre attraverso funzioni in python (ad esempio il fatto che un abbonato non possa partecipare a due sedute all'interno dello stesso slot).
Il motivo per cui è stata scelta una via piuttosto che l'altra è dovuta dalla necessità o meno di riportare dei feedback lato front end per l'utente.

Integrità

Vincoli check

Tabella utenti

```
datanascita DATE CHECK (datanascita < CURRENT_DATE)
```

Controllo che la data di nascita sia minore della data corrente.

Tabella abbonamenti

```
costo REAL CHECK(costo > 0)
```

Controllo che il costo dell'abbonamento sia maggiore di 0.

Tabella abbonati

```
abbonamento INT NOT NULL,
durata INT CHECK(durata > 0),
CHECK (datafineabbonamento > datainizioabbonamento)
```

Controllo che la chiave esterna abbonamento non sia NULL.
Controllo che la durata in giorni dell'abbonamento sia maggiore di 0.
Controllo che la data di fine abbonamento sia maggiore della data di inizio abbonamento.

Tabella stanze

```
dimensione INT CHECK(dimensione > 0)
```

Controllo che la dimensione della stanza sia maggiore di 0.

Tabella salapesi

```
dimensione INT CHECK(dimensione > 0),
```

```
iscrittimax INT CHECK(iscrittimax > 0),
```
Controllo che la dimensione della sala sia maggiore di 0.
Controllo che il numero di iscritti massimi della sala sia maggiore di 0.

Tabella corsi
```
iscrittimax INT CHECK(iscrittimax > 0),
istruttore INT NOT NULL,
stanza INT NOT NULL,
```
Controllo che il numero di iscritti massimi della sala sia maggiore di 0.
Controllo che la chiave esterna istruttore non sia NULL.
Controllo che la chiave esterna stanza non sia NULL.

Tabella sedutecorsi
```
corso INT NOT NULL,
```
Controllo che la chiave esterna corso non sia NULL.

Tabella sedutesalepesi
```
salapesi INT NOT NULL,
```
Controllo che la chiave esterna salapesi non sia NULL.

Tabella slot
```
personemax INT CHECK(personemax > 0),
CHECK (orafine > orainizio)
```
Controllo che il numero di persone massime prenotabili nello slot sia maggiore di 0.
Controllo che l'ora di fine dello slot sia minore dell'ora di inizio dello slot.

Tabella infomazioni
```
accessisettimana INT CHECK(accessisettimana > 0),
slotgiorno INT CHECK(slotgiorno > 0 AND slotgiorno < 6),
personemaxslot INT CHECK(personemaxslot > 0),
personemq INT CHECK(personemq > 0),
```
Controllo che il numero di accessi alla settimana sia maggiore di 0.
Controllo che il numero di slot prenotabili al giorno sia compreso tra 0 e 6.
Controllo che il numero di persone massime prenotabili in uno slot sia maggiore di 0.
Controllo che il numero di metri quadri garantiti a persona sia maggiore di 0.

Tabella controlli
```
controllo INT CHECK(controllo >= 0),
```
Controllo che il numero di controlli sia maggiore o uguale a 0.

Trigger e funzioni

Procedure aggiungi_corsi_slot

```
DROP PROCEDURE IF EXISTS aggiungi_corsi_slot(idcorso INT, giorno INT, slot TIME);
CREATE PROCEDURE aggiungi_corsi_slot(idcorso INT, giorno INT, slot TIME) AS $$
    DECLARE giorno_finale DATE = CURRENT_DATE + 30;
    DECLARE giorno_buffer DATE = CURRENT_DATE + 1;
    DECLARE giorno_settimana INT;
    DECLARE slot_c INT;
    DECLARE seduta_c INT;
    DECLARE cs_iscrittimax INT;
    BEGIN
        SELECT COALESCE(MAX(id),0) INTO seduta_c FROM sedutecorsi;
        SELECT iscrittimax INTO cs_iscrittimax FROM corsi where id=idcorso;
        WHILE giorno_buffer < giorno_finale LOOP
            SELECT EXTRACT(DOW FROM giorno_buffer) INTO giorno_settimana;
            IF (giorno_settimana = giorno) THEN
                SELECT s.id INTO slot_c FROM slot s WHERE s.giorno = giorno_buffer AND s.orainizio = slot;
                INSERT INTO corsislot VALUES (idcorso, slot_c, cs_iscrittimax);
                seduta_c = seduta_c + 1;
                INSERT INTO sedutecorsi(id, corso, dataseduta) VALUES(seduta_c, idcorso, giorno_buffer+slot);
                giorno_buffer = giorno_buffer + 7;
            ELSE
                giorno_buffer = giorno_buffer + 1;
            END IF;
        END LOOP;
    END;
$$ LANGUAGE 'plpgsql';
```

Al momento della creazione di un nuovo corso, si occupa di inserire nelle tabelle corsislot e sedutecorsi gli id del corso e dello slot, che inizia all'ora indicata dal parametro "slot", per ogni giorno della settimana indicato dal parametro "giorno", per ogni settimana, per i prossimi 30 giorni.

Procedure aggiungi_salapesi_slot

```
DROP PROCEDURE IF EXISTS aggiungi_salapesi_slot(idsala INT);
CREATE PROCEDURE aggiungi_salapesi_slot(idsala INT) AS $$
    DECLARE giorno_finale DATE = CURRENT_DATE + 30;
    DECLARE giorno_buffer DATE = CURRENT_DATE + 1;
    DECLARE slot_sp INT;
    DECLARE seduta_sp INT;
    DECLARE sp_iscrittimax INT;
    DECLARE slot_s TIME = '05:30:00';
    DECLARE s slot%rowtype;
    BEGIN
        SELECT COALESCE(MAX(id),0) INTO seduta_sp FROM sedutesalepesi;
        SELECT iscrittimax INTO sp_iscrittimax FROM salepesi where id=idsala;
        WHILE giorno_buffer <= giorno_finale LOOP
            FOR s IN SELECT * FROM slot WHERE giorno = giorno_buffer LOOP
                INSERT INTO salapesislot VALUES (idsala, s.id, sp_iscrittimax);
                seduta_sp = seduta_sp + 1;
                INSERT INTO sedutesalepesi(id, salapesi, dataseduta) VALUES(seduta_sp, idsala, giorno_buffer+s.orainizio);
            END LOOP;
            giorno_buffer = giorno_buffer + 1;
        END LOOP;
    END;
$$ LANGUAGE 'plpgsql';
```

Al momento della creazione di una nuova sala pesi, si occupa di inserire nelle tabelle salapesislot e sedutesalepesi gli id della sala pesi e di ogni slot appartenente ai prossimi 30 giorni, perchè abbiamo assunto che le sale pesi siano sempre aperte.

Trigger trigger_persone_mq

```
DROP TRIGGER IF EXISTS t_personemq ON informazioni CASCADE;
DROP FUNCTION IF EXISTS trigger_personemq();
CREATE FUNCTION trigger_personemq() RETURNS trigger AS $$
    BEGIN
        UPDATE corsi c SET iscrittimax = (SELECT s.dimensione
                                          FROM stanze s
                                          WHERE c.stanza=s.id) / NEW.personemq;
        UPDATE salepesi sp SET iscrittimax = (sp.dimensione) / NEW.personemq;
        UPDATE slot s SET personemax = 1 + (SELECT COALESCE(SUM(c.iscrittimax),0)
                                    FROM corsi c JOIN corsislot cs ON c.id=cs.corso
                                    WHERE s.id=cs.slot)
                                    +
                                    (SELECT COALESCE(SUM(sp.iscrittimax),0)
                                    FROM salepesi sp JOIN salapesislot ss ON sp.id=ss.salapesi
                                    WHERE s.id=ss.slot)
                        WHERE giorno > CURRENT_DATE + 7;
        UPDATE salapesislot ps SET iscrittimax = (SELECT sp.iscrittimax FROM salepesi sp WHERE sp.id=ps.salapesi)
                               WHERE slot IN (SELECT id FROM slot WHERE giorno > CURRENT_DATE + 7);
        UPDATE corsislot cs SET iscrittimax = (SELECT c.iscrittimax FROM corsi c WHERE c.id=cs.corso)
                               WHERE slot IN (SELECT id FROM slot WHERE giorno > CURRENT_DATE + 7);
        UPDATE informazioni SET personemaxslot = (SELECT COALESCE(MAX(personemax),0) FROM slot WHERE giorno > CURRENT_DATE + 7);
        RETURN NULL;
    END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER t_personemq AFTER UPDATE OF personemq ON informazioni
FOR EACH ROW
EXECUTE FUNCTION trigger_personemq();
```

Sistema per ogni corso e sala pesi il numero massimo di persone che possono accedervi in base alla dimensione dello spazio di allenamento e al numero di metri quadri minimi da garantire ad ogni persona.
Imposta il numero di persone massime all'interno di ogni slot come la somma degli iscritti massimi possibili dei corsi e delle sale pesi presenti in quello slot.

Trigger trigger_iscrittimax_corso_stanza

```
DROP TRIGGER IF EXISTS t_iscrittimax_corso_stanza ON corsi CASCADE;
DROP FUNCTION IF EXISTS trigger_iscrittimax_corso_stanza();
CREATE FUNCTION trigger_iscrittimax_corso_stanza() RETURNS trigger AS $$
    DECLARE pmq INT;
    BEGIN
        SELECT personemq INTO pmq FROM informazioni;
        UPDATE corsi c SET iscrittimax = (SELECT s.dimensione
                                          FROM stanze s
                                          WHERE c.stanza=s.id) / pmq;
        RETURN NULL;
    END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER t_iscrittimax_corso_stanza AFTER INSERT ON corsi
FOR EACH ROW
EXECUTE FUNCTION trigger_iscrittimax_corso_stanza();
```

Sistema per ogni nuovo corso, dopo il suo inserimento, il numero massimo di iscritti che può avere in base alla stanza in cui si svolge.

Trigger trigger_personemaxslot

```
DROP TRIGGER IF EXISTS t_personemaxslot ON informazioni CASCADE;
DROP FUNCTION IF EXISTS trigger_personemaxslot();
CREATE FUNCTION trigger_personemaxslot() RETURNS trigger AS $$
    DECLARE valori INT;
    DECLARE totale INT;
    DECLARE totale_corsi INT;
    DECLARE totale_sale INT;
    DECLARE s slot%rowtype;
    DECLARE corso_r corsi%rowtype;
    DECLARE sala_r salepesi%rowtype;
    BEGIN
        UPDATE slot sl SET personemax = 1 + (SELECT COALESCE(SUM(iscrittimax),0)
                                    FROM corsislot WHERE slot=sl.id)
                                    +
                                    (SELECT COALESCE(SUM(iscrittimax),0)
                                    FROM salapesislot WHERE slot=sl.id)
                    WHERE giorno >= CURRENT_DATE AND giorno <= CURRENT_DATE + 7;
        FOR s IN SELECT * FROM slot WHERE giorno > CURRENT_DATE + 7 ORDER BY giorno LOOP
            SELECT COUNT(DISTINCT(c.corso)) INTO totale_corsi FROM corsislot c WHERE c.slot = s.id;
            SELECT COUNT(DISTINCT(sp.salapesi)) INTO totale_sale FROM salapesislot sp WHERE sp.slot = s.id;
            totale = totale_corsi + totale_sale;
            IF (totale = 0) THEN
                totale = 1;
            END IF;
            valori = NEW.personemaxslot / totale;
            UPDATE corsislot SET iscrittimax = valori WHERE slot=s.id AND iscrittimax > 1 AND iscrittimax > valori;
            UPDATE salapesislot SET iscrittimax = valori WHERE slot=s.id AND iscrittimax > 1 AND iscrittimax > valori;
            UPDATE slot SET personemax = 1 + (SELECT COALESCE(SUM(iscrittimax),0)
                                    FROM corsislot WHERE slot=s.id)
                                    +
                                    (SELECT COALESCE(SUM(iscrittimax),0)
                                    FROM salapesislot WHERE slot=s.id)
                    WHERE id=s.id;
        END LOOP;

        RETURN NEW;
    END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER t_personemaxslot BEFORE UPDATE OF personemaxslot ON informazioni
FOR EACH ROW
EXECUTE FUNCTION trigger_personemaxslot();
```

Si occupa di gestire il caso in cui si voglia mettere un limite al numero massimo di persone che possono accedere ad uno slot e abbassa il numero di persone iscrivibili ai corsi/sale di ciascuno slot.

Trigger trigger_username_diversi_utenti

```
DROP TRIGGER IF EXISTS t_username_diversi_utenti ON utenti CASCADE;
DROP FUNCTION IF EXISTS trigger_username_diversi_utenti();
CREATE FUNCTION trigger_username_diversi_utenti() RETURNS trigger AS $$
    BEGIN
        IF (EXISTS (SELECT * FROM utenti u WHERE u.username=NEW.username OR u.email=NEW.email)) THEN
            RETURN NULL;
        END IF;
        RETURN NEW;
    END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER t_username_diversi_utenti BEFORE INSERT OR UPDATE ON utenti
FOR EACH ROW
EXECUTE FUNCTION trigger_username_diversi_utenti();
```

Serve per far si che non esistano due utenti con lo stesso username o con la stessa email, questo è utile anche per motivi di sicurezza dato che le password uguali appaiono come diverse quando vengono cifrate e inserite all'interno del database.

Trigger trigger_insert_clienti

```
DROP TRIGGER IF EXISTS t_insert_nonabbonati ON nonabbonati CASCADE;
DROP TRIGGER IF EXISTS t_insert_abbonati ON abbonati CASCADE;
DROP FUNCTION IF EXISTS trigger_insert_clienti();
CREATE FUNCTION trigger_insert_clienti() RETURNS trigger AS $$
    BEGIN
        IF (NOT EXISTS (SELECT * FROM clienti c WHERE c.id=NEW.id)) THEN
            RETURN NULL;
        END IF;
        RETURN NEW;
    END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER t_insert_abbonati BEFORE INSERT OR UPDATE ON abbonati
FOR EACH ROW
EXECUTE FUNCTION trigger_insert_clienti();

CREATE TRIGGER t_insert_nonabbonati BEFORE INSERT OR UPDATE ON nonabbonati
FOR EACH ROW
EXECUTE FUNCTION trigger_insert_clienti();
```

Serve per far si che vengano inseriti all'interno delle tabelle abbonati e non abbonati solo utenti che sono effettivamente clienti della palestra.

Procedure check_genera_giorno_e_slot

```
DROP PROCEDURE IF EXISTS check_genera_giorno_e_slot();
CREATE PROCEDURE check_genera_giorno_e_slot() AS $$
    DECLARE pms INT;
    DECLARE idSlot INT;
    DECLARE giorno_tmp DATE;
    DECLARE oraIn TIME = '05:30:00';
    DECLARE oraFin TIME = '08:50:00';
    BEGIN
        SELECT personemaxslot INTO pms FROM informazioni;
        SELECT MAX(id) INTO idSlot FROM slot;
        idSlot = idSlot + 1;
        FOR i IN 1..30 LOOP
            giorno_tmp = CURRENT_DATE + i;
            IF (NOT EXISTS(SELECT * FROM giorni WHERE data = giorno_tmp)) THEN
                INSERT INTO giorni VALUES(giorno_tmp);
                FOR j IN 0..5 LOOP
                    INSERT INTO slot VALUES(idSlot, pms, giorno_tmp, oraIn, oraFin);
                    idSlot = idSlot + 1;
                    oraIn = oraFin + '00:10:00';
                    oraFin = oraIn + '02:50:00';
                END LOOP;
                oraIn = '05:30:00';
                oraFin = '08:50:00';
            END IF;
        END LOOP;
    END;
$$ LANGUAGE 'plpgsql';
```

Si occupa della creazione dei giorni con i relativi slot prenotabili per i prossimi 30 giorni, controllando che tutti o in parte non siano già presenti.

Procedure check_aggiungi_corsi_slot

```
DROP PROCEDURE IF EXISTS check_aggiungi_corsi_slot();
CREATE PROCEDURE check_aggiungi_corsi_slot() AS $$
    DECLARE idSedute INT;
    DECLARE idSlot INT;
    DECLARE c corsi%rowtype;
    DECLARE data_tmp TIMESTAMP;
    DECLARE data_buff TIMESTAMP;
    BEGIN
        SELECT COALESCE(MAX(id),0) INTO idSedute FROM sedutecorsi;
        idSedute = idSedute + 1;
        FOR c IN SELECT * FROM corsi ORDER BY id LOOP
            FOR i IN 0..6 LOOP
                SELECT se.dataseduta INTO data_tmp FROM sedutecorsi se WHERE se.corso=c.id AND i=(
                    SELECT EXTRACT(DOW FROM se.dataseduta));
                IF (data_tmp IS NOT NULL) THEN
                    FOR j IN 0..3 LOOP
                        IF (NOT EXISTS(SELECT * FROM sedutecorsi se WHERE se.corso=c.id AND se.dataseduta=(data_tmp + (
                            INTERVAL '1 day' * 7*j)))) THEN
                            INSERT INTO sedutecorsi VALUES(idSedute, c.id, data_tmp + (INTERVAL '1 day' * 7*j));
                            SELECT dataseduta INTO data_buff FROM sedutecorsi WHERE id=idSedute;
                            SELECT id INTO idSlot FROM slot WHERE giorno=(data_buff::date) AND orainizio=(data_buff::time);
                            IF (NOT EXISTS (SELECT * FROM corsislot WHERE corso=c.id AND slot=idSlot)) THEN
                                INSERT INTO corsislot VALUES(c.id, idSlot, c.iscrittimax);
                            END IF;
                            idSedute = idSedute + 1;
                        END IF;
                    END LOOP;
                END IF;
            END LOOP;
        END LOOP;
    END;
$$ LANGUAGE 'plpgsql';
```

Si occupa di aggiornare per ogni corso le relative sedute, gestendo direttamente anche la tabella corsislot.

Procedure check_aggiungi_salepesi_slot

```
DROP PROCEDURE IF EXISTS check_aggiungi_salepesi_slot();
CREATE PROCEDURE check_aggiungi_salepesi_slot() AS $$
    DECLARE idSedute INT;
    DECLARE sp salepesi%rowtype;
    DECLARE g giorni%rowtype;
    DECLARE s slot%rowtype;
    BEGIN
        SELECT COALESCE(MAX(id),0) INTO idSedute FROM sedutesalepesi;
        idSedute = idSedute + 1;
        FOR sp IN SELECT * FROM salepesi ORDER BY id LOOP
            FOR g IN SELECT * FROM giorni ORDER BY data LOOP
                FOR s IN SELECT * FROM slot WHERE giorno=g.data LOOP
                    IF (NOT EXISTS(SELECT * FROM sedutesalepesi se WHERE se.salapesi=sp.id AND se.dataseduta=(
                        s.giorno + s.orainizio))) THEN
                        INSERT INTO sedutesalepesi VALUES(idSedute, sp.id, s.giorno + s.orainizio);
                        IF (NOT EXISTS (SELECT * FROM salapesislot WHERE salapesi=sp.id AND slot=s.id)) THEN
                            INSERT INTO salapesislot VALUES(sp.id, s.id, sp.iscrittimax);
                        END IF;
                        idSedute = idSedute + 1;
                    END IF;
                END LOOP;
            END LOOP;
        END LOOP;
    END;
$$ LANGUAGE 'plpgsql';
```

Si occupa di aggiornare per ogni salapesi le relative sedute, gestendo direttamente anche la tabella salapesislot.

Trigger trigger_check_data_fine_abbonamento

```
DROP TRIGGER IF EXISTS check_data_fine_abbonamento ON controlli CASCADE;
DROP FUNCTION IF EXISTS trigger_check_data_fine_abbonamento();
CREATE FUNCTION trigger_check_data_fine_abbonamento() RETURNS trigger AS $$
    DECLARE personemaxslot_buffer INT;
    BEGIN
        DELETE FROM giorni WHERE data < CURRENT_DATE - 7;
        INSERT INTO nonabbonati (id) SELECT id FROM abbonati WHERE datafineabbonamento < CURRENT_DATE;
        INSERT INTO prenotazioninonabbonati (nonabbonato, slot) SELECT p.abbonato, p.slot
                                                                FROM prenotazioni p JOIN slot sl ON p.slot=sl.id
                                                                WHERE p.abbonato IN (SELECT * FROM nonabbonati) AND
                                                                sl.giorno >= CURRENT_DATE - 7;
        DELETE FROM abbonati WHERE datafineabbonamento < CURRENT_DATE;
        CALL check_genera_giorno_e_slot();
        CALL check_aggiungi_corsi_slot();
        CALL check_aggiungi_salepesi_slot();
        SELECT personemaxslot INTO personemaxslot_buffer FROM informazioni;
        UPDATE informazioni SET personemaxslot = personemaxslot_buffer;
        RETURN NEW;
    END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER check_data_fine_abbonamento BEFORE UPDATE ON controlli
FOR EACH ROW
EXECUTE FUNCTION trigger_check_data_fine_abbonamento();
```

Si occupa di mantenere il database aggiornato per quanto riguarda gli utenti abbonati, non abbonati, i giorni, gli slot, le sedute dei corsi e delle sale, e la possibilità di prenotare le sedute future.

Trigger trigger_cancella_prenotazioni_corsi

```
DROP TRIGGER IF EXISTS cancella_prenotazioni_corsi ON corsi CASCADE;
DROP FUNCTION IF EXISTS trigger_cancella_prenotazioni_corsi();
CREATE FUNCTION trigger_cancella_prenotazioni_corsi() RETURNS trigger AS $$
    DECLARE sc sedutecorsi%rowtype;
    DECLARE da_eliminare TIMESTAMP;
    DECLARE c INT;
    BEGIN
        SELECT id INTO c FROM corsi WHERE id=OLD.id;
        FOR sc IN SELECT * FROM sedutecorsi WHERE corso=c AND dataseduta > CURRENT_DATE LOOP
            da_eliminare = sc.dataseduta;
            DELETE FROM prenotazioni WHERE slot = (
                SELECT id FROM SLOT WHERE giorno = da_eliminare::date AND orainizio = da_eliminare::time)
                                        AND abbonato IN (SELECT abbonato FROM abbonatisedutecorsi WHERE seduta = sc.id);
        END LOOP;
        RETURN OLD;
    END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER cancella_prenotazioni_corsi BEFORE DELETE ON corsi
FOR EACH ROW
EXECUTE FUNCTION trigger_cancella_prenotazioni_corsi();
```

Nella rimozione di un corso si occupa di rimuovere tutte le prenotazioni effettuate da tutti gli abbonati iscritti alle sedute future del corso che sta per essere eliminato.

Trigger trigger_cancella_prenotazioni_salepesi

```
DROP TRIGGER IF EXISTS cancella_prenotazioni_salepesi ON salepesi CASCADE;
DROP FUNCTION IF EXISTS trigger_cancella_prenotazioni_salepesi();
CREATE FUNCTION trigger_cancella_prenotazioni_salepesi() RETURNS trigger AS $$
    DECLARE sc sedutesalepesi%rowtype;
    DECLARE s INT;
    BEGIN
        SELECT id INTO s FROM salepesi WHERE id=OLD.id;
        FOR sc IN SELECT * FROM sedutesalepesi WHERE salapesi=s AND dataseduta > CURRENT_DATE LOOP
            DELETE FROM prenotazioni WHERE slot IN (SELECT id FROM SLOT WHERE giorno > CURRENT_DATE)
                                       AND abbonato IN (SELECT abbonato FROM abbonatisedutesalepesi WHERE seduta = sc.id);
        END LOOP;
        RETURN OLD;
    END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER cancella_prenotazioni_salepesi BEFORE DELETE ON salepesi
FOR EACH ROW
EXECUTE FUNCTION trigger_cancella_prenotazioni_salepesi();
```

Nella rimozione di una sala pesi si occupa di rimuovere tutte le prenotazioni effettuate da tutti gli abbonati iscritti alle sedute future della sala che sta per essere eliminata.

Trigger trigger_cancella_stanza_occupata

```
DROP TRIGGER IF EXISTS cancella_stanza_occupata ON stanze CASCADE;
DROP FUNCTION IF EXISTS trigger_cancella_stanza_occupata();
CREATE FUNCTION trigger_cancella_stanza_occupata() RETURNS trigger AS $$
    BEGIN
        IF (EXISTS (SELECT * FROM corsi WHERE stanza=OLD.id)) THEN
            RETURN NULL;
        END IF;
        RETURN OLD;
    END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER cancella_stanza_occupata BEFORE DELETE ON stanze
FOR EACH ROW
EXECUTE FUNCTION trigger_cancella_stanza_occupata();
```

Controlla che prima di rimuovere una stanza non sia utilizzata da nessun corso.

Trigger trigger_cancella_istruttore_occupato

```
DROP TRIGGER IF EXISTS cancella_istruttore_occupato ON utenti CASCADE;
DROP FUNCTION IF EXISTS trigger_cancella_istruttore_occupato();
CREATE FUNCTION trigger_cancella_istruttore_occupato() RETURNS trigger AS $$
    BEGIN
        IF (EXISTS (SELECT * FROM corsi WHERE istruttore=OLD.id)) THEN
            RETURN NULL;
        END IF;
        RETURN OLD;
    END;
$$ LANGUAGE 'plpgsql';

CREATE TRIGGER cancella_istruttore_occupato BEFORE DELETE ON utenti
FOR EACH ROW
EXECUTE FUNCTION trigger_cancella_istruttore_occupato();
```

Controlla che prima di rimuovere un istruttore non sia responsabile di nessun corso.

Performance

Indici

Sono stati definiti degli indici per migliorare e rendere più efficienti le query, principalmente sulle chiavi e sulle chiavi esterne (per semplificare le join).

Esempi di indice su chiave

```
CREATE INDEX ON utenti (id);
CREATE INDEX ON abbonatisedutecorsi (seduta, abbonato);
```

Esempi di indice su chiave esterna

```
CREATE INDEX ON corsi (stanza);
CREATE INDEX ON slot (giorno);
```

Sicurezza e astrazione DBMS

É stato utilizzato SQLAlchemy ORM in modo da effettuare un'astrazione delle query dal linguaggio SQL, garantendo così anche maggiore sicurezza.
L'utilizzo di ORM permette di effettuare più controlli sul tipaggio e sul passaggio di parametri, inoltre operazioni come inserimenti, cancellazioni, join, ecc... sono gestiti automaticamente e dunque rendono il programma meno error prone.

È stato creato un decoratore di nome @admin_required utile per tutte quelle funzioni nelle varie root dove solo l'amministratore è autorizzato ad accedere.
L'uso di un decoratore permette il riutilizzo del codice all'interno del file e diminuisce la probabilità di commettere errori sintattici da parte del programmatore.
Si presta utile anche perchè centralizza la sezione di codice da modificare in un'unica funzione, e dunque semplifica il suo mantenimento con eventuale futura modifica.

Molte root sono state marcate come @login_required in modo che venga controllato automaticamente che l'utente sia loggato prima che provi ad accedervi.
Inoltre nelle operazioni significative vengono effettuati sempre dei controlli preventivi prima di eseguire le loro istruzioni (ad esempio viene controllato che l'utente sia l'admin prima che si acceda a zone sensibili)
Al momento della registrazione di un utente, alla sua password viene concatenato il suo indirizzo email (unico per definizione e controllato da un trigger che sia tale) e il server ne calcola l'MD5. In questo modo persino password uguali, grazie al fatto che le mail sono diverse, generano MD5 diversi, garantendo un livello di sicurezza maggiore in caso di intrusione nella base di dati.
Ad ogni tentativo di login queste operazioni vengono svolte nuovamente con i dati inviati dal client, il server si occupa di confrontare i risultati e se combaciano allora l'utente è autenticato.
Grazie a questa tecnica non serve che il server salvi le password in chiaro.

Ulteriori informazioni

Eseguire il progetto con:

```
set FLASK_APP=project.py
```

seguito da:

```
flask run
```

I file SQL contenenti la creazione delle tabelle, dei triggers e della popolazione iniziale sono contenuti nella cartella "pgadmin_file".
Oltre a creare le tabelle della base di dati ed ad inserire i vari triggers e le relative funzioni in pgadmin, permettono di aggiungere, tra le varie cose, alcune informazioni, abbonamenti e stanze utili per avere qualche dato iniziare per l'utilizzo dell'applicazione.
Consigliamo di aggiungere immediatamente dei corsi, con i relativi istruttori, e delle sale pesi in modo da poter avere delle attività a cui i clienti potranno iscriversi.

Inoltre bisognerebbe eseguire i trigger della pagina di amministrazione in questo ordine:

1. Controlli giornalieri, meno utile al primo avvio perchè i dati vengono tutti generati alla generazione delle varie attività ma diventa fondamentale in futuro per mantenere le diverse funzionalità attive
2. Modifica mq minimi, in modo che automaticamente si generino dei limiti dentro i quali è garantita la sicurezza per gli utenti della palestra
3. Modifica persone massime slot, se si vuole inserire un numero diverso da quello generato precedentemente in automatico

Gli altri due trigger riguardanti il numero massimo di accessi giornalieri e settimanali possono essere eseguiti in ordine qualsiasi.