# Documentazione Progetto BASI DI DATI
# Finestra Dicembre 2021

# Gruppo

- Furlan Alessandro 854723
- Babato Thomas  875469

# Tema Scelto

Creazione App per gestione palestra in tempo di COVID

# Tecnologie Utilizzate

Le seguenti tecnologie sono state utilizzate per la creazione dell'app,alcune di queste verrano approfondite più avanti nella Documentaziona

- Python
- Sql-Alchemy
- Flask
- Flask Login
- Flask BCrypt
- PostGreSQL
- Docker

# Installazione Applicazione

La directory aggiornata dell'app è disponibile all'indirizzo https://github.com/Haka91/ProgettoBasi2021

## Requisiti di Sistema

Per garantire il funzionamento dell'applicazione su tutti i SO sono stati ideati 2 container Docker gestiti tramite le funzionalità di Docker Compose.
Per questo è necessario installare sulla propria macchina Docker(indifferente se la versione Desktop o solo Console)

## Step installazione:

1. Una volta scaricato Docker e la Directory aprire la console di comando(CMD/BASH) nella cartella Services
2. Eseguire la seguente linea di Comando:
    a. Se Windows **docker compose build**
    b. Se Unix/Linux/MacOs **docker-compose build**
3. Attendere l'installazione di tutte le dependencies,alla fine eseguire:
    a. Se Windows **docker compose up**
    b. Se Unix/Linux/MacOs **docker-compose up**
4. A questo punto l'app sarà attiva,attendere la conferma dalla riga di comando che tutti e due i container sono in running e andare all'indirizzo **Localhost** di un qualsiasi browser per utilizzare l'app

## Troubleshooting

I container sono in running, ma l'app non parte.

Stoppare il container principale e poi avviare manualmente prima i Db Services e poi i Web Services

Non posso buildare i container,errore porta già occupata.

Alcuni sistemi operativi o programmi potrebbero utilizzare le porte inserite nella configurazione dei container,in quel caso aprire il file **docker-compose.yml** e modificare il valore a destra della porta specificata in ports che da il problema

# Progetto

## Tema assegnatoci

L'emergenza Covid-19 richiede accessi contingentati alle palestre, quindi avete deciso di sviluppare un sistema di controllo degli accessi tramite prenotazioni online. L'applicazione deve presentare un calendario suddiviso in un numero di slot prenotabili e deve essere possibile imporre un limite al numero di accessi per ciascuno slot. Oltre alla classica sala pesi, la palestra offre anche una serie di corsi, ciascuno dei quali è gestito da un istruttore e può avere un numero massimo di iscritti.

# Requisiti Progetto:

Il progetto richiede come minimo lo svolgimento dei seguenti punti:

1. 1. Progettazione concettuale e logica dello schema della base di dati su cui si appoggerà all'applicazione, opportunamente commentata e documentata.
2. 2. Creazione di un database, anche artificiale, tramite l'utilizzo di uno specifico DBMS. La creazione delle tabella e l'inserimento dei dati può essere effettuato anche con uno script esterno al progetto.
3. 3. Implementazione di un front-end minimale basato su HTML e CSS. E' possibile utilizzare framework CSS esistenti come W3.CSS, Bootstrap o altri. E' inoltre possibile fare uso di JavaScript per migliorare l'esperienza utente, ma non è strettamente necessario e non influisce sulla valutazione finale.
4. 4. Implementazione di un back-end basato su Flask e SQLAlchemy (o Flask-SQLAlchemy). Per migliorare il progetto e la relativa valutazione `e raccomandato gestire anche i seguenti aspetti:
    a. 1. Integrità: definizione di vincoli e trigger per garantire l'integrità dei dati gestiti dall'applicazione.
    b. 2. Sicurezza: definizione di opportuni ruoli e politiche di autorizzazione, oltre che di ulteriori meccanismi atti a migliorare il livello di sicurezza dell'applicazione (es. difese contro SQL injection).
    c. 3. Performance: definizione di indici o viste materializzate sulla base delle query più frequenti previste.
    d. 4. Transazioni: utilizzo di transazioni per garantire la correttezza di funzionalità strutturate.
    e. 5. Astrazione dal DBMS sottostante: uso di Expression Language o ORM per astrarre dal dialetto SQL.

E' possibile focalizzarsi solo su un sottoinsieme di questi aspetti, ma i progetti eccellenti cercheranno di coprirli tutti ad un qualche livello di dettaglio. E' meglio approfondire adeguatamente solo alcuni di questi aspetti piuttosto che coprirli tutti in modo insoddisfacente.

# Requisiti 1-2

## Modello Concettuale tema

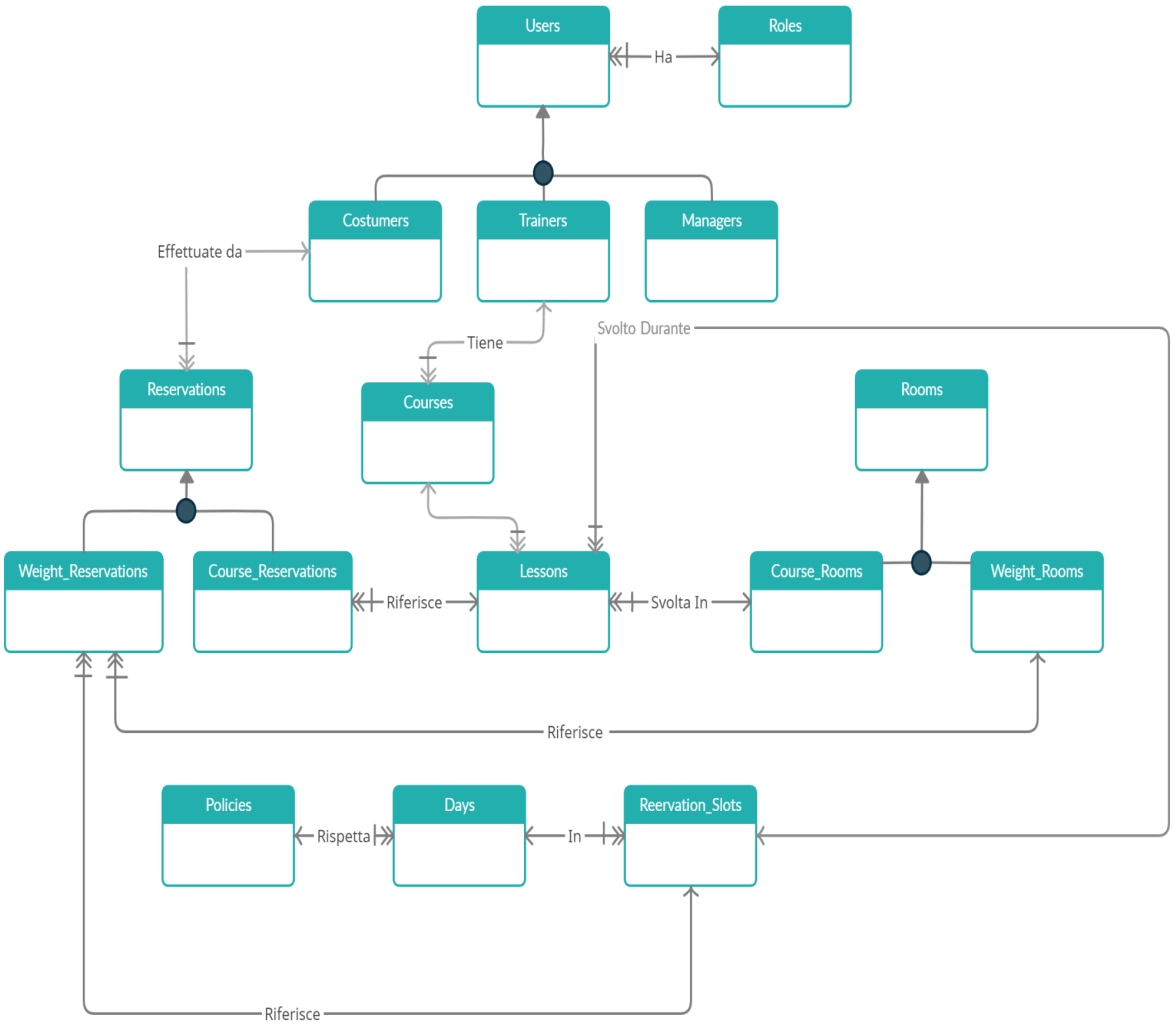
Prima di iniziare lo sviluppo dell'applicazione e della base sottostante ,dato il tema assegnatoci abbiamo dovuto definire nello specifico il modello concettuale del nostro DB completando nei dettagli la traccia fornita dal professore:

La nostra app dovrà dare la possibilità a degli utenti di prenotare degli slot orari nelle stanze pesi della palestra,che avranno o meno un limite massimo di presenze per slot,ma rispettando comunque la capienza massima della sala stessa.
Oltre a questo i nostri utenti possono iscriversi a delle lezioni di alcuni corsi tenuti da degli utenti-trainer in stanze specifiche per corsi

Gli utenti trainer terranno dei corsi personali per cui potranno creare delle lezioni alle quali gli utenti potranno iscriversi.I corsi verranno assegnati loro da un utente-manager(questo per evitare che un  trainer possa crearsi 9999 corsi senza il consenso del manager)
L'utente manager avrà il compito , oltre che di creare i corsi , di gestire gli orari di apertura e di chiusura della palestra,e di gestire le policy associate a tali giorni.
Una policy dovrà specificare la capienza massima in percentuale delle stanze nella giornata e le prenotazioni massime che un utente potrà fare nella giornata(questo per la sala pesi,le lezioni saranno escluse da questo vincolo per scelta)
La policy sarà applicata all'intero giorno e ovviamente a cascata a tutti gli slot che comporranno tale giorno

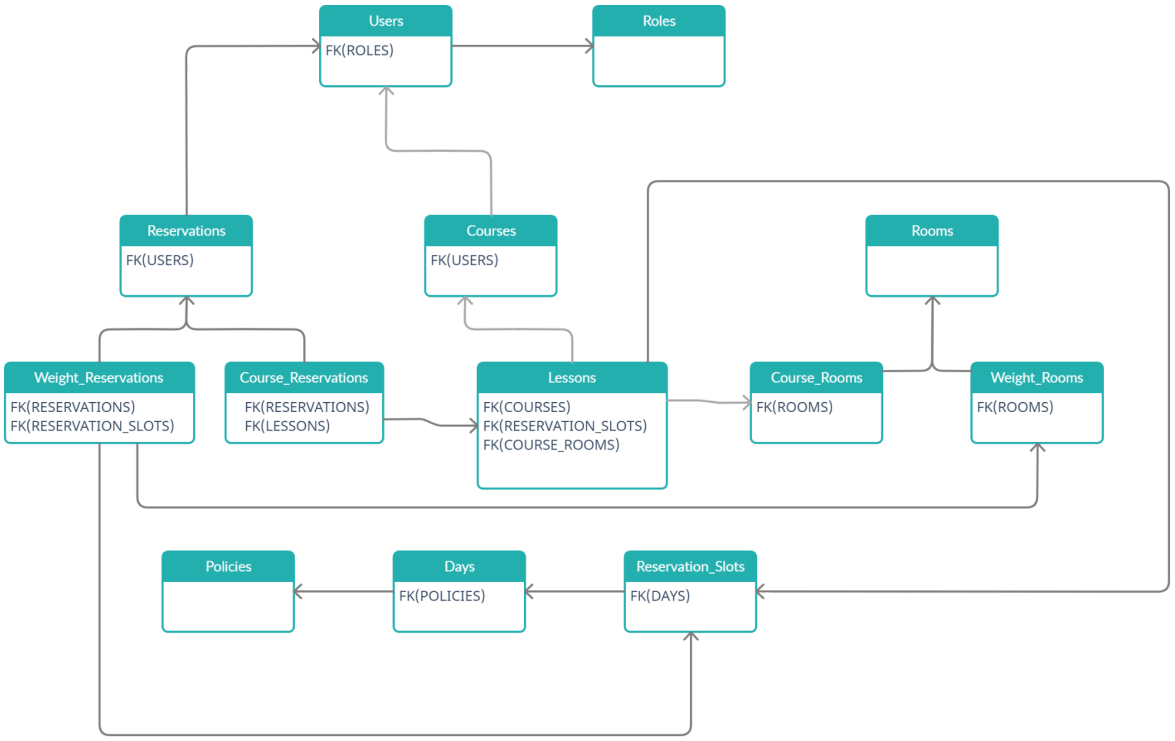
## Modello Concettuale Schema

A seguito di tale specifica possiamo creare quindi lo schema concettuale con le relative entità.
**Attenzione,lo schema per semplicità di lettura è mancante degli attributi,tali attributi saranno poi specificati nella specifica delle tabelle del modello logico.**




## Modello Logico

A seguito della creazione del modello Concettuale la trasformazione in modello Logico risulta semplice,anche perché siamo riusciti a mantenere relazioni di tipo N:1(molti a uno) su tutto lo schema questo ci da la possibilità di tradurlo quasi in maniera uguale senza bisogno di creare tabelle di relazione N:N(molti a molti).
**Attenzione lo scherma per semplicità di lettura è mancante degli attributi(tranne per le FK),la specifica di ogni singola tabella sarà presente nella prossima sezione**




Si può notare come sia stato scelto di unificare le tabelle Costumers-Trainers-Managers a favore di una singola tabella Users,questo ci da la possibilità di generalizzare le funzionalità dell'applicazione senza vincolare la creazione di ruoli specifici per esigenze specifiche
es.Un trainer potrebbe voler iscriversi ad una lezione di altri trainers o fare prenotazioni per le sale pesi,oppure un manager potrebbe tenere corsi  tutti casi che accadono spesso nella realtà.
**Questo però crea una piccola anomalia nei dati, cioè un trainer o manager potrebbe iscriversi alle sue stesse lezioni.**
Dato il tipo di anomalia non grave (ne possibile stando alla buona fede dei trainers) abbiamo scelto comunque di mantenere questo schema dati i benefici che comporta.

## Modello Logico Specifiche Tabelle

### Prefazione

Dato che questa applicazione e il relativo  DB sono stati creati con l'intento di sfruttare appieno le funzionalità di Sql-Alchemy ORM ,nella creazione delle tabelle sono presenti campi non relativi alla "pura" tabella,ma che sono necessari per il corretto mantenimento del DB.
Per lo stesso motivo pensando questa applicazione per un più ampio spettro di Db possibili **non sono presenti triggers specifici del db** (anche perchè oramai i triggers sono in buona parte superati,proprio per dare spazio ad una più ampia liberta al momento della creazione dell'applicazione).

**Ciononostante** non significa che non siano fatti controlli per garantire la sicurezza e la correttezza dei dati,solo sono stati portati a livello di Applicazione o Classe.

Courses

```
class Course(Base):

    __tablename__='Courses'


    id= Column("id",Integer, primary_key=True)
    name = Column(String(50),unique=True)
    description=Column(String(50))
    trainer = Column(Integer,ForeignKey("Users.id"),nullable=False)
    isvisible = Column(Boolean,default=False)
    maxcostumers=   Column(Integer,CheckConstraint('maxcostumers<101'),
    CheckConstraint('maxcostumers>0'),nullable=False,default=100)


    lessons_obj= relationship("Lesson",back_populates="course_obj",
    cascade="all, delete")
    trainer_obj= relationship("User", back_populates="courses_obj")
```

**Campi tabella**

- id=primary key
- name=nome del corso,ha vincolo unique
- description=descrizione corso
- trainer=Fk che rimanda allo user che sarà il propietario del corso
- isvisible=campo bool che imposta la visibilità del corso sulla nostra app sia per i trainer sia per gli user,è sempre visibile per la gestione del manager
- maxcostumers= iscritti massimi ad una lezione del corso,globale per tutte le lezioni,
  check constraint: un corso deve avere la possibilità di avere almeno un iscritto alle lezioni e un massimo di 100

**Relationship(Sql Alchemy-ORM)**

- lessons_obj= lista di lezioni che sono associate al corso
- trainer_obj= oggetto di tipo User che fa riferimento alla FK trainer

**“Trigger-Funzioni”**

```
    def isDeletable(self):
        deletable=True
        if len(self.lessons_obj)==0:
```

```
            return deletable
        else:
            for lesson in self.lessons_obj:
               if(lesson.reservation_slot_obj.day >(date.today()-
timedelta(days=30))):
                    deletable=False
        return deletable
```

Se sono presenti lezioni fino a 30 giorni fa il corso non è cancellabile

Days

```
class Day(Base):


    __tablename__='Days'
    id=Column(Integer,primary_key=True)
    date = Column(Date,nullable=False,unique=True)
    opening=Column(DateTime,nullable=False)
    closing=Column(DateTime,nullable=False)
    break_time =Column(DateTime,nullable=True)
    break_slot  =Column(Integer,CheckConstraint('break_slot>=0')
    ,CheckConstraint('break_slot<47')
    ,nullable=False,default=0)
    policy=Column(Integer,ForeignKey("Policies.id"),nullable=False,default=1)


    reservation_slots_obj=relationship("Reservation_Slot",back_populates="day_obj",
    cascade="all, delete")
    policy_obj=relationship("Policy",back_populates="days_obj")
```

**Campi Tabella**

- id= primary key
- date= specifica il giorno ,ha vincolo unique
- opening= specifica l'apertura
- closing= specifica la chiusura
- break_time= specifica un eventuale pausa,può essere null
- break_slot= specifica la durata in slot della pausa,check costraint= la durata non può superare la giornata(i timeslot sono espressi in mezz'ora 24*2=48)
- policy= FK che rimanda alla policy applicata al giorno

**Relationship(Sql Alchemy-ORM)**

- reservation slots_obj= lista di reservation slots associati al giorno
- policy_obj= oggetto di tipo Policy che fa riferimento alla FK policy

**“Trigger-Funzioni”**

```
    def is_deletable(self):

        if(self.date<(date.today()-timedelta(days=30))):
            return True
        else:
            for reservationSlot in self.reservation_slots_obj:
                if (len(reservationSlot.lessons_obj)>0 or
            len(reservationSlot.weight_reservations_obj)>0):
                    return False
            return True
```

Se sono presenti lezioni o prenotazioni in un qualsiasi Reservation_slot fino a 30 giorni fa il giorno non è cancellabile

Lessons

```
class Lesson(Base):

    __tablename__='Lessons'


    id = Column("id",Integer, primary_key=True)
    reservation_slot=Column(Integer,ForeignKey("Reservation_slots.id")
```

```
    ,nullable=False)
    course = Column(Integer,ForeignKey("Courses.id"),nullable=False)
    course_room = Column(Integer,ForeignKey("Course_Rooms.id"),nullable=False)


    course_room_obj=relationship("Course_Room",back_populates="lessons_obj")
    course_obj =  relationship("Course",back_populates="lessons_obj")

    reservation_slot_obj=relationship("Reservation_Slot"
    ,back_populates="lessons_obj")
    course_reservations_obj=relationship("Course_Reservation"
    ,back_populates="lesson_obj", cascade="all, delete")


    __table_args__ = (UniqueConstraint('reservation_slot', 'course_room'
   , name='singlelessoninaroom'),
                      )
```

**Campi Tabella**

- id= primary key
- reservation_slot= Fk che fa riferimento all Reservation_slot occupato dalla lezione
- course = Fk che fa riferimento al corso padre della lezione
- course_room= Fk che fa riferimento alla stanza in cui si tiene il corso

**Relationship(Sql Alchemy-ORM)**

- course_room_obj=oggetto di tipo Course_Room che fa riferimento alla FK course_room
- course_obj=oggetto di tipo Course che fa riferimento alla FK course
- reservation_slot_obj=oggetto di tipo reservation_slot che fa riferimento alla FK reservation_slot
- course_reservations_obj=lista di oggetti Course_Reservation associati alla lezione

**Table Costraint**

- singlelessoninaroom= vincolo che controlla che non ci sia più di una lezione sulla coppia slot/stanza

**“Trigger-Funzioni”**

```
    def is_deletable(self):

        return len(self.course_reservations_obj)==0
```

Se sono presenti prenotazioni in questa lezione non posso cancellarla

Policies

```
class Policy(Base):

    __tablename__='Policies'


    id = Column("id",Integer, primary_key=True)
    name = Column(String(50),unique=True)
    room_percent=Column(Integer,CheckConstraint('room_percent<101')
    ,CheckConstraint('room_percent>0'),nullable=False,default=100)
    max_user_reserv=Column(Integer,CheckConstraint('max_user_reserv>0')
    ,CheckConstraint('max_user_reserv<49'),nullable=False,default=48)

    days_obj=relationship("Day",back_populates="policy_obj", cascade="all, delete")
```

**Campi Tabella**

- id= primary key
- name= nome della policy,vincolo Unique
- room_percent=vincolo che impedisce che la percentuale di occupazione sia inferiore allo 0% e superiore al 100%

**Relationship(Sql Alchemy-ORM)**

- days_obj=lista di oggetti Days associati alla Policy

**“Trigger-Funzioni”**

```
    def is_deletable(self):

        if(len(self.days_obj)==0):
            return True
        else:
            for day in self.days_obj:

                if(day.date>(date.today()-timedelta(days=30))):
                    for reservationSlot in day.reservation_slots_obj:
```

```
                        if (len(reservationSlot.lessons_obj)>0 or
len(reservationSlot.weight_reservations_obj)>0):
                            return False
            return True
```

Se sono presenti lezioni o prenotazioni più recenti di 30 giorni o se la policy ha meno di 30 giorni non posso cancellarla

Reservation_Slot

```
class Reservation_Slot(Base):

    __tablename__='Reservation_slots'


    id = Column("id",Integer,primary_key=True)
    slot_time =Column(Time,nullable=False)
    #non fa riferimento alla Pk di days,ma ad una colonna unique,questo ci evita un
passaggio in più nelle query orm per il controllo della data
    day = Column(Date,ForeignKey("Days.date"),nullable=False)

    lessons_obj=relationship("Lesson",back_populates="reservation_slot_obj",
    cascade="all, delete")
    weight_reservations_obj=relationship("Weight_Room_Reservation"
    ,back_populates="reservation_slot_obj", cascade="all, delete")
    day_obj=relationship("Day", back_populates="reservation_slots_obj")
```

**Campi Tabella**

- id= primary key
- slot_time= indica l'ora e il minuto occupato dallo slot,non nullable
-

**Relationship(Sql Alchemy-ORM)**

- days_obj= lista di oggetti Days associati allo Slot
- lessons_obj= lista di oggetti Lesson associati allo Slot
- weight_reservations_obj= lista di oggetti Weight_Reservations associati allo Slot

Reservations

```
class Reservation(Base):

    __tablename__='Reservations'


    id = Column("id",Integer, primary_key=True)
    is_weight=Column(Boolean)
    user = Column(Integer,ForeignKey("Users.id"),nullable=False)


    user_obj=relationship("User",back_populates="reservations_obj")

    __mapper_args__={
      'polymorphic_on':is_weight,
    }


class Weight_Room_Reservation(Reservation):

        __tablename__='Weight_Room_Reservations'

        id = Column(Integer,ForeignKey(Reservation.id),primary_key=True)

        weight_room = Column(Integer,ForeignKey("Weight_Rooms.id"),nullable=False)
        reservation_slot=Column(Integer,ForeignKey("Reservation_slots.id")
             ,nullable=False)

        reservation_slot_obj=relationship("Reservation_Slot"
            ,back_populates="weight_reservations_obj")
        weight_room_obj=relationship("Weight_Room"
            ,back_populates="weight_reservations_obj")

        __mapper_args__ = {
        'polymorphic_identity':True,
```

```
    }




class Course_Reservation(Reservation):

        __tablename__='Course_Reservations'

        id = Column(Integer,ForeignKey(Reservation.id),primary_key=True)
        lesson = Column(Integer,ForeignKey("Lessons.id"),nullable=False)

        lesson_obj=relationship("Lesson",back_populates="course_reservations_obj")



        __mapper_args__ = {
        'polymorphic_identity':False,

    }
```

Il caso della tabella Reservations,così come per la tabella Rooms è particolare,infatti usiamo un meccanismo interno a Sql-Alchemy  chiamato **Joined Table Inheritance** che ci permette di far ereditare a delle sottoclassi la classe principale mantenendo però un mapping  specifico sulle sottoclassi e classi della tabella,questo ci permette di effettuare query sulle sottoclassi, nonostante a livello di dati nel Db è presente solo una tabella con un descrittore specificato nella classe principale

```
'polymorphic_on'
```

che permette di distinguere le tabelle

**Campi Tabella**

- id= primary key
- is weight = descrittore che ci permette di scegliere tra Weight_Room_Reservation e Course_Reservation
- user= Fk alla tabella Users
- weight_room= Fk alla tabella Rooms
- reservation_slot= Fk alla tabella Reservation_Slots
- lesson= Fk alla tabella Lessons

**Relationship(Sql Alchemy-ORM)**

- user_obj= oggetto di tipo User che fa riferimento alla FK user
- weight_room_obj= oggetto di tipo Weight_Room che fa riferimento alla FK weight_room
- reservation_slot_obj= oggetto di tipo Reservation_Slot che fa riferimento alla FK reservation_slot
- lesson_obj= oggetto di tipo Lesson che fa riferimento alla FK lesson

Roles

```
class Role(Base):

    __tablename__='Roles'

    id = Column("id",Integer, primary_key=True)
    name = Column(String(50),unique=True)

    users_obj=relationship("User",back_populates="role_obj", cascade="all, delete")
```

**Campi Tabella**

- id= primary key
- name= nome del ruolo,vincolo di unique

**Relationship(Sql Alchemy-ORM)**

- users_obj= lista di oggetti User associati a Role

Rooms

```
class Room(Base):
    __tablename__='Rooms'


    id = Column("id",Integer, primary_key=True)
    name = Column(String(50),nullable=False,unique=True)
    description=Column(String(200),nullable=False)
    max_capacity=Column(Integer,CheckConstraint('max_capacity>0'),nullable=False)
    is_weight=Column(Boolean)
    isvisible = Column(Boolean,default=True)

    __mapper_args__={
```

```
      'polymorphic_on':is_weight
    }

class Weight_Room(Room):

    __tablename__='Weight_Rooms'

    id = Column(Integer,ForeignKey(Room.id),primary_key=True)

    weight_reservations_obj=relationship("Weight_Room_Reservation"
     ,back_populates="weight_room_obj",cascade="all, delete")

    __mapper_args__ = {

        'polymorphic_identity':True
    }

class Course_Room(Room):

    __tablename__='Course_Rooms'

    id = Column(Integer,ForeignKey(Room.id),primary_key=True)

    lessons_obj=relationship("Lesson",back_populates="course_room_obj"
      ,cascade="all, delete")

    __mapper_args__ = {

        'polymorphic_identity':False
    }
```

Il caso della tabella Rooms,così come per la tabella Reservations è particolare,infatti usiamo un meccanismo interno a Sql-Alchemy  chiamato **Joined Table Inheritance** che ci permette di far ereditare a delle sottoclassi la classe principale mantenendo però un mapping  specifico sulle sottoclassi e classi della tabella,questo ci permette di effettuare query sulle sottoclassi, nonostante a livello di dati nel Db è presente solo una tabella con un descrittore specificato nella classe principale

```
'polymorphic_on'
```

che permette di distinguere le tabelle

**Campi Tabella**

- id= primary key
- name= nome stanza
- description= descrizione stanza
- max_capacity= capienza massima della stanza,non può essere minore di 0

- is weight = descrittore che ci permette di scegliere tra Weight_Room e Course_Room
- isvisible=campo bool che imposta la visibilità della stanza sulla nostra app sia per i trainer sia per gli user,è sempre visibile per la gestione del manager

**Relationship(Sql Alchemy-ORM)**

- weight_reservations_obj= lista di oggetti Weight_Reservations associati a Room
- lessons_obj= lista di oggetti Lesson associati a Room

**“Triggers-Funzioni”**

```
    def is_deletable(self):
        deletable=True
        if(len(self.weight_reservations_obj)==0):
            return deletable
        else:
            #se le prenotazioni sono più vecchie di 30 giorni posso cancellarle
            for weightReserations in self.weight_reservations_obj:
               if( weightReserations.reservation_slot_obj.day >(date.today()-
timedelta(days=30))):
                   deletable=False
        return deletable
```

```
    def is_deletable(self):
        deletable=True
        if(len(self.lessons_obj)==0):
            return deletable
        else:
            #se le lezioni sono più vecchie di 30 giorni posso cancellarle
            for lesson in self.lessons_obj:
               if( lesson.reservation_slot_obj.day >(date.today()-
timedelta(days=30))):
                   deletable=False
        return deletable
```

Se sono presenti lezioni o prenotazioni più recenti di 30 giorni non posso cancellare le stanze

Users

```
class User(UserMixin,Base):

    __tablename__='Users'
```

```
    id = Column("id",Integer, primary_key=True)
    name = Column(String(50), nullable=False)
    surname = Column( String(50), nullable=False)
    email = Column(String(40), nullable=False,unique=True)
    cellular=Column(String(40), nullable=False)
    address = Column( String(30), nullable=False)
    city = Column(String(30), nullable=False)
    password = Column( String, nullable=False)
    role = Column(Integer,ForeignKey("Roles.id"),nullable=False,default=3)



    role_obj=relationship("Role",back_populates="users_obj")

    courses_obj=relationship("Course"
      ,back_populates="trainer_obj",cascade="all, delete")
    reservations_obj=relationship("Reservation"
      ,back_populates="user_obj",cascade="all, delete")
```

**Campi Tabella**

- id= primary key
- name= nome user
- surname= cognome user
- email= email user,usata per le procedure di login,non può essere Null ed ha vincolo Unique
- cellular= telefono user
- address= indirizzo user
- city=città user
- password= password user,la password è salvata tramite Flask-Bcrypt e non in chiaro,maggiori informazioni nella sezione sicurezza progetto
- role= Fk che fa riferimento a Roles

**Relationship(Sql Alchemy-ORM)**

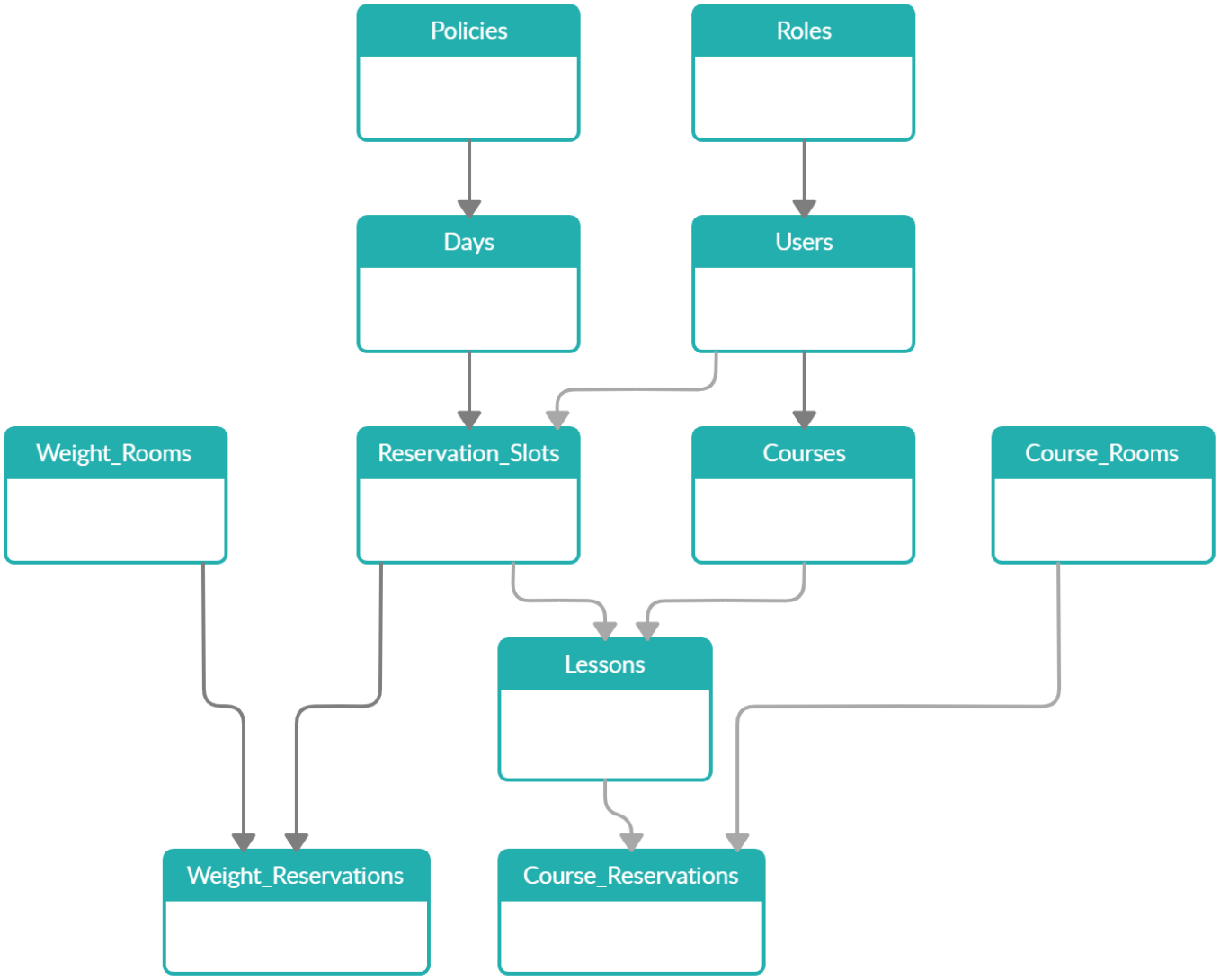
- courses_obj= lista di oggetti Course associati a User
- reservations_obj= lista di oggetti Reservation associati a User
- role_obj= oggetto di tipo Role che fa riferimento alla FK role

## Policy di Delete

Come si può notare dagli schemi sopra riportati,non sono state impostate delle politiche di delete sulle FK,ma sono state impostate a livello di Relationship,questo perchè l'engine di Sql-Alchemy crea in maniera automatica delle politiche di delete sfruttando le relationship(il default è on delete NULL),

La nostra architettura del DB si è basata sul Delete Cascade rappresentata nello schema sottostante

**Dall'alto si effettua il Delete a cascata**




# Requisito 3

## Frontend

Lo sviluppo del frontend si è basato sull'utilizzo esclusivo dell'html e del css.
Per lo sviluppo dell'interfaccia si è preso spunto da esempi di utilizzo delle librerie Bootstrap.
La versione di che abbiamo utilizzato è la 4.5.2.

## Ereditarietà

Per evitare il ripetersi delle stesse righe di codice e risparmiarci tempo sullo sviluppo abbiamo sfruttato la funzionalità di ereditarietà dei template. Con questa funzionalità offertaci dal template engine Ninja2 abbiamo

creato il file base.html, situato dentro la cartella Templates.
Il suddetto file contiene al suo interno la struttura base di tutte le pagine web del progetto.
Questo file, viene ereditato da tutti gli altri file .html del progetto.
Il riempimento del file base.html, da parte di tutte le altre pagine html che lo estendono, avviene tramite l'utilizzo di tue tag:

- {% block navbar %} che contiene i link che verranno visualizzati sulla navbar;
- {% block body%} che contiene il corpo della pagina html.

### Suddivisione

I diversi template, quindi le diverse pagine html, sono state suddivise in varie cartelle. La suddivisione è stata attuata anche in base all'area riservata a cui appartengono.
In totale abbiamo 4 cartelle:

- General: cartella nella quale troviamo le schermate di login, registrati e homepage;
- Instructor: pagine dell'area riservata dell'istruttore;
- Manager: pagine dell'area riservata del gestore;
- User: pagine dell'area riservata dell'utente.

### Blueprint

Il nostro utilizzo delle Blueprint è stato spinto dalla necessità di suddividere il software in componenti distinti. Questo comporta degli indubbi vantaggi:

- maggior chiarezza del codice;
- miglior mantenimento del progetto;
- minor tempo speso per la ricerca e correzione di bugs.

La nostra implementazione delle blueprint è stata fatta basandoci sulla suddivisione dell'applicazione web in 4 aree distinte, tre che corrispondono alle tre aree riservate (user, instructor e manager) e la quarta che è riferita agli utenti non autenticati.
Tale suddivisione è possibile notarla anche nella divisione fisica dei file .html, presenti dentro la cartella Templates.

## Requisito 4

Come già specificato in diversi punti della documentazione il Back-End è stato sviluppato oltre che con Flask e Sql-Alchemy(CORE e ORM) con Flask Login e Flask BCrypt.Ora andremo a vedere nello specifico per cosa sono state utilizzate tali tecnologie

## Flask e Sql-Alchemy

Queste 2 tecnologie sono le principali tecnologie (oltre ad essere obbligatorie) su cui si basa il progetto, Sfruttare Sql-Alchemy ORM ci ha permesso di rendere più generale possibile la nostra applicazione,in modo che possa essere utilizzata su praticamente qualsiasi DB,questo grazie al fatto che Sql-Alchemy va a "conversare" con il Db tramite un suo particolare interprete(nel caso di PostgreSQL `psycopg2`) e quindi se mai un giorno volessimo cambiare Db ci basterà cambiare poche parti del codice perché tutto funzioni in maniera uguale a prima(nel caso specifico di questa applicazione andrebbe cambiata anche la creazione dello user che è l'unica cosa scritta in puro SQL,ma in una vera applicazione la creazione del Db sarebbe esterna).

In tutta l'applicazione non è presente(tranne come detto nella creazione dello user di connessione) una singola riga di SQL,questo grazie ai meccanismi di relationship impostati nelle classi che compongono il nostro DB,questo ha reso molto semplice effettuare Query molto complicate in "poche righe" mantenendo un ottima leggibilità del codice.

Vediamo ad esempio la funzione di contact Tracing,**personalmente**(Furlan Alessandro),non credo sarei riuscito a trasformare in query la funzione sottostante senza perderci interi giorni

```
    def contactTracing(self):
        #lo dichiaro nella funzione per evitare un possibile circular input
        from Models.Reservations import Weight_Room_Reservation
        usersListWithDuplicates=list()
        #mi aggiungo per essere sicuro di essere nella lista da rimuovere alla fine
        usersListWithDuplicates.append(self)
        for reservation in self.reservations_obj:
            if(reservation.is_weight):
                #recupero tutte le prenotazioni fatte nello stesso slot,nella stessa
ora,nella stessa stanza,tra 7 giorni fa e ieri
                reservationsForRoomAndSlot=
session.query(Weight_Room_Reservation).filter(Weight_Room_Reservation.weight_room==r
eservation.weight_room and
Weight_Room_Reservation.reservation_slot==reservation.reservation_slot and
Weight_Room_Reservation.reservation_slot_obj.day<datetime.today().date() and
Weight_Room_Reservation.reservation_slot_obj.day>(datetime.today()-timedelta(days=7)
).date())
                for weightReservation in reservationsForRoomAndSlot:
                    usersListWithDuplicates.append(weightReservation.user_obj)
            else:
                #la lezione è stata fatta tra 7 giorni fa e ieri?
                if
reservation.lesson_obj.reservation_slot_obj.day<(datetime.today().date()) and
reservation.lesson_obj.reservation_slot_obj.day>(datetime.today()-timedelta(days=7))
.date() :
```

```
                    #anche il trainer viene a contatto quindi lo riprendo dal corso
della lezione

usersListWithDuplicates.append(reservation.lesson_obj.course_obj.trainer_obj)
                    for courseReservation in
reservation.lesson_obj.course_reservations_obj:
                        usersListWithDuplicates.append(courseReservation.user_obj)
        #un utente potrebbe essere un trainer oppure potrebbe essere stato un
trainer ed ora è uno user,controllo se ha mai tenuto corsi(l'impossibilità di
cancellare i corsi prima di 30 gg mi garantisce il ritrovamento di essi)
        for course in self.courses_obj:
            for lesson in course.lessons_obj:
                #lezione tenuta tra 7 giorni fa e ieri?
                if lesson.reservation_slot_obj.day<(datetime.today().date()) and
lesson.reservation_slot_obj.day>(datetime.today()-timedelta(days=7)).date() :
                    for personalCourseReservation in lesson.course_reservations_obj:

usersListWithDuplicates.append(personalCourseReservation.user_obj)

        #ora ho una lista piena di duplicati probabilmente che sistemo con il set
__eq__ ed __hash__ sono già integrati da flask ed sqlalchemy

        userlistWithNoDuplicate=list(set(usersListWithDuplicates))
        #rimuovo me stesso dai miei contatti
        userlistWithNoDuplicate.remove(self)

        return tuple(userlistWithNoDuplicate)
```

**Bisogna stare però molto attenti**,mentre da una parte questo meccanismo di ORM aumenta la semplicità del codice scritto,se non perfettamente implementato completo di opzioni sulle query potrebbe portare ad un degrado rapido di prestazioni sulla scalabilità del progetto.

La nostra applicazione su questo aspetto, essendo un prototipo non ha pensato alla scalabilità come requisito fondamentale

## Flask-Login

Un altra parte molto importante del progetto è data dalla libreria Flask Login,flask login ci permette di creare delle sessioni a livello di users per autenticarsi nella nostra app in modo da garantirne il corretto funzionamento.Tramite una chiave segreta da noi impostata e ereditando la classe UserMixin (nella nostra applicazione è a livello di Environment ) crea dei cookie che (a detta degli sviluppatori di Flask-Login) sono

tamper proof,questo ci permette di semplificare notevolmente le procedure di login/logout dell'user e implementa anche un ottima funzionalità di remember me(persistenza della sessione per un dato tempo anche se l'user chiude l'app,**attenzione,**se viene effettuato il logout la sessione viene persa).
Bisogna però ricordarsi che la sessione non viene generata a livello di Db,ma a livello di utente,questo significa che se vengono fatte delle modifiche allo user(ad eccezione della cancellazione dello user stesso) esse non si rifletteranno sulla sessione dello user se precedentemente aperta(es. cambio il ruolo di uno user da trainer ad user).
Per ovviare a questo si potrebbe inserire un token comprensivo di versione dell'utente inviato con ogni richiesta,oppure salvare le sessioni attive dello user sul db e validarle di volta in volta,oppure forzare un limite massimo alla "vita" di una sessione.
Per motivi di tempo e di complessità noi abbiamo scelto la terza ipotesi,anche perchè al momento la funzionalità di remember me risulta avere dei problemi con alcuni browser tramite il comando

```
app.permanent_session_lifetime =timedelta(minutes=30)
```

### Flask-BCrypt

Ultima ma non meno importante è la libreria Flask-BCrypt,tale libreria ci permette di Hashare le password sul nostro Db e di renderle sicure anche nel caso fossero rubate da un agente malevolo.
Infatti,mentre altre librerie si prefissano di trovare metodi di hash sempre più veloci,BCrypt è pensato per essere il più lento possibile,questo rende **MOLTO** difficile e lento un tentativo di recupero della password in chiaro se non si è a conoscenza del salt e hash usato.
Un altro vantaggio di BCrypt è che non dobbiamo impostare noi un salt,dato che una volta bindato all'app tramite

```
bcrypt = Bcrypt(app)
```

creerà e gestirà lui dei salt appropriati.

## Requisiti Aggiuntivi

### Integrità

Per garantire una corretta politica di integrità sul nostro Db oltre ad usare i vincoli a livello di tabella sono state generate una serie di funzioni/sanitizzazioni dei dati che sono visionabili nel codice.Infatti come specificato non usiamo i triggers per non vincolare buona parte della nostra app ad uno specifico Db.

Oltre a questo è stata pensata una "policy" di conservazione dati molto stringente dato che il tema prevedeva la gestione di una palestra ai tempi del COVID e che è stato implementato un meccanismo di Contact Tracing.

Come si può vedere nelle Tabelle specifiche del requisito 2,molti record,seppur supportati da una buona politica di cascade **non possono essere cancellati prima di 30gg** se non

vuoti(Days,Policies,Rooms,Lessons),questo onde evitare la possibile perdita di informazione a livello di Contact Tracing(es. un istruttore cancella una lezione di 2 giorni fa e un partecipante risulta positivo etc etc)

Oltre a ciò questa scelta “estrema” è dovuta al fatto che non è stato implementato un modo **in quest’app** di comunicare ad uno user un'eventuale cancellazione della sua prenotazione.
Sappiamo che è possibile gestire anche un server mail tramite flask-login,ma le nostre competenze ed il tempo a disposizione ci hanno fatto scegliere a favore di una decisione semplice e giusta,anche se non completamente condivisibile.
Per mitigare questo però sono state inserite una serie di aggiunte sulla visibilità di corsi  e lezioni per aiutare un eventuale manager nella gestione.

## Sicurezza

Oltre alle tecnologie già utilizzate e spiegate nei capitoli precedenti per aumentare la sicurezza della nostra app/Db sono state aggiunte altre funzionalità quali.

### Decoratori di Endpoint

Tutti gli endpoint dell’app sono stati coperti da delle funzioni che ne limitano l’accesso a specifici User/ruoli. questi decoratori sono

```
@login_required
@at_least_user_required
@at_least_trainer_required
@at_least_manager_required
```

mentre @login_required viene implementato da flask-login ed è a noi trasparente,gli altri 3 decoratori sono stati impostati da noi e sono così definiti

```
def at_least_manager_required(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        if not current_user.is_anonymous:
            if current_user.get_role() == 3 :
                return f(*args, **kwargs)
            else:
                flash("ERROR 404 PAGE NOT FOUND","error")

                return redirect(url_for("general.index"))
        else:
            flash("ERROR 404 PAGE NOT FOUND","error")
            return redirect(url_for("general.index"))
```

```
    return wrapper
```

```
def at_least_trainer_required(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        if not current_user.is_anonymous:
            if current_user.get_role() >= 2:
                return f(*args, **kwargs)
            else:
                flash("ERROR 404 PAGE NOT FOUND","error")
                return redirect(url_for("general.index"))
        else:
            flash("ERROR 404 PAGE NOT FOUND","error")
            return redirect(url_for("general.index"))

    return wrapper
```

```
def at_least_user_required(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        if not current_user.is_anonymous:
            if current_user.get_role() >= 1:
                return f(*args, **kwargs)
            else:
                flash("ERROR 404 PAGE NOT FOUND","error")
                return redirect(url_for("general.index"))
        else:
            flash("ERROR 404 PAGE NOT FOUND","error")
            return redirect(url_for("general.index"))

    return wrapper
```

Come si può vedere dai corpi delle funzioni tutte le funzioni sono pensate per lavorare all'unisono con la nostra politica di accesso gerarchico dell'applicazione.(Admin>Trainer>User).

Anche per questo si è scelto di implementare a livello di user,un singolo user generico che abbia i poteri di **Select,Insert,Update,Delete** su tutte le tabelle(tranne Roles,su quella è presente solo il **Select**) oltre che di **Usage,Select on ALL Sequences** (necessari all incremento delle pk).
Lo user viene creato dopo la creazione del DB e perciò non ha poteri di Drop sulle Tabelle non essendo l'owner.

## Performance

Come detto in precedenza non abbiamo concepito l'app con in mente la scalabilità e performance e perciò tutte le funzioni di query sono standard come da implementazione di Sql-Alchemy,non sono state pensate paging di dati.

## Transazioni

Sfruttando le session di SqlAlchemy tutta l'app si basa su transazioni(per quanto banali),quindi se per un motivo qualsiasi un add o un delete non dovessero andare a buon fine la transazione verrebbe eliminata e sarebbe fatto un rollback allo stato precedente.
Un buon esempio di Transazione inpiù livelli è l'aggiunta di Days,essi infatti sono gli unici creatori di Reservation_Slots e perciò se anche solo uno slot non potesse essere inserito tutti gli slot precedenti e il Days stesso subirebbero un rollback.

Add a livello di Days

```
   def add_obj(self):
        from Models.Reservation_Slots import Reservation_Slot
        try:
            if(self.opening>=self.closing):
                flash("Orario di chiusura precedente a apertura")
                return False
            if(self.break_time is not None and self.opening>=self.break_time ):
                flash("Orario di break precedente ad apertura")
                return False
            session.add(self)
            guard=True
            startTime=self.opening
            #creo per ogni giorno i relativi timeslot
            while(startTime<=self.closing and guard):
                if(self.break_time==None ):
                    tempReservation=Reservation_Slot(startTime,self.date)
                    #se ci sono errori cancello il commit della sessione
                    guard=tempReservation.add_obj()
                    startTime=startTime + timedelta(minutes=30)
                elif((self.break_time +
timedelta(minutes=(self.break_slot*30)))<self.closing):
                      #controllo che la pausa non sia più lunga della chiusura,nel
caso cancello tutto
                     while(startTime<=self.closing and guard):
```

```
                        if(startTime<self.break_time or startTime>(self.break_time
+timedelta(minutes=(self.break_slot*30)))):

tempReservation=Reservation_Slot(startTime,self.date)
                                guard=tempReservation.add_obj()


                        startTime=startTime + timedelta(minutes=30)
                else:
                    flash("break più lungo dell'apertura della palestra")
                    session.rollback()
                    return False

            session.commit()
            return True
        except Exception as e:
             #questo errore lo vediamo solo lato admin se mai dovesse
succedere,possiamo considerare il manager un utente "sicuro" per mostrargli errori
specifici
            if " duplicate key value violates unique constraint" in e.__str__():
                flash("Uno o piu' giorno già esistenti nel DB")
            else:
                flash(e)
            session.rollback()
            return False
```

Add a livello di Slot

```
   def add_obj(self):

        try:
            session.add(self)
            #non aggiungo direttamente da qui il reservation slot perchè li aggiungo
da Day
            return True
        except Exception as e:
            print(e)
            session.rollback()
            return False
```

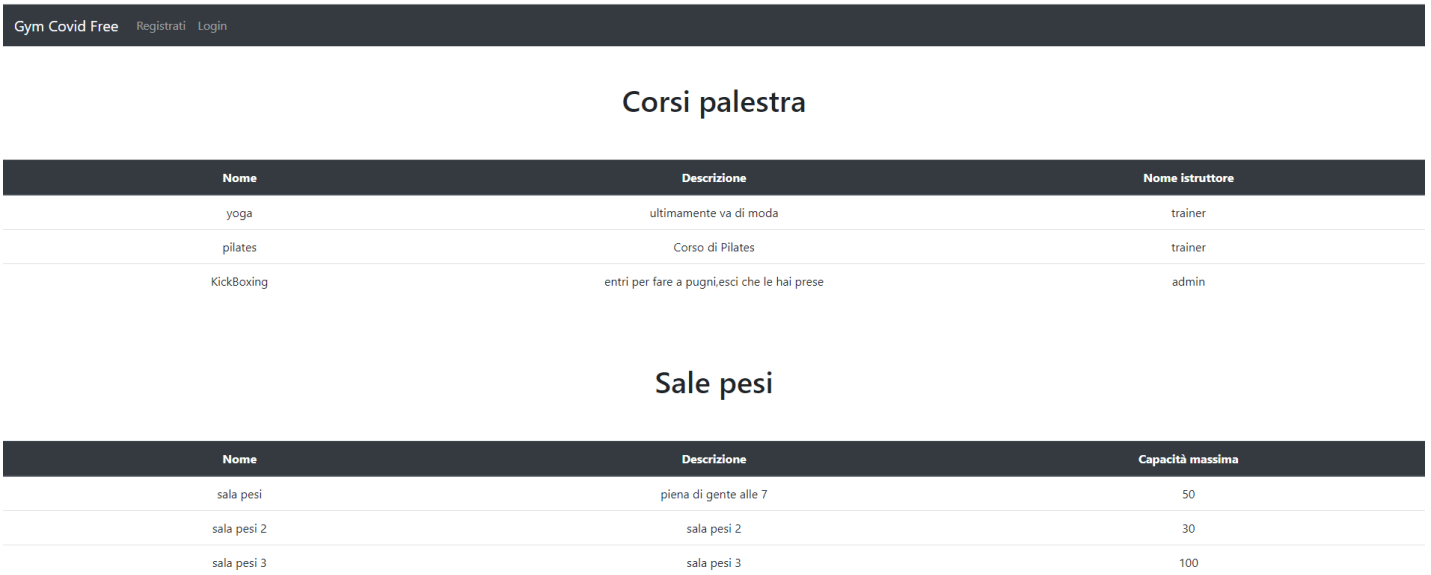
## Astrazione dal DBMS sottostante

Come già dimostrato in precedenza e come visibile dal codice TUTTA l'app si basa sull'utilizzo di SQL-ALCHEMY ORM .

# Presentazione dell' app e relative funzionalità

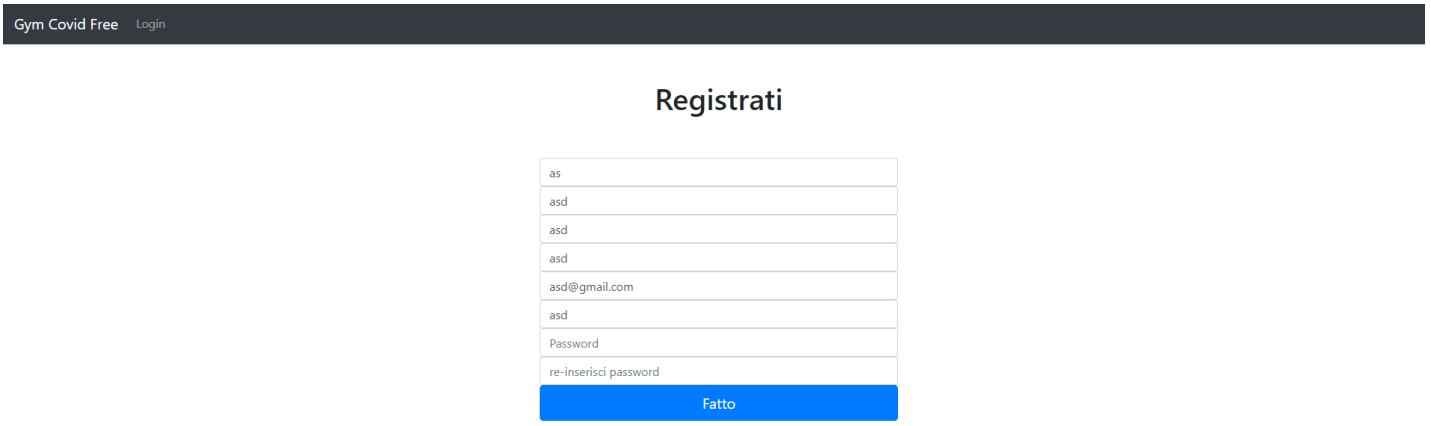
Nelle prossime pagine vi guideremo passo passo nelle funzionalità dell'app tramite una navigazione guidata in essa.
Si possono usare 3 account per visitare l'app.
admin@gmail.com admin
trainer@gmail.com trainer
costumer@gmail.com costumer

## Homepage



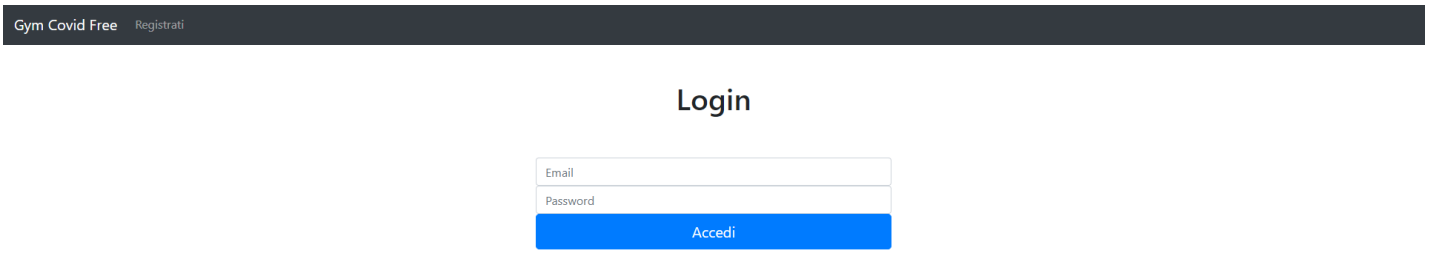

Dall'homepage un user non registrato potrà vedere i corsi che vengono tenuti in palestra e le relative sale pesi e corsi,da qui potrà scegliere se fare login nell'account oppure se registrarsi alla palestra

# Registrati



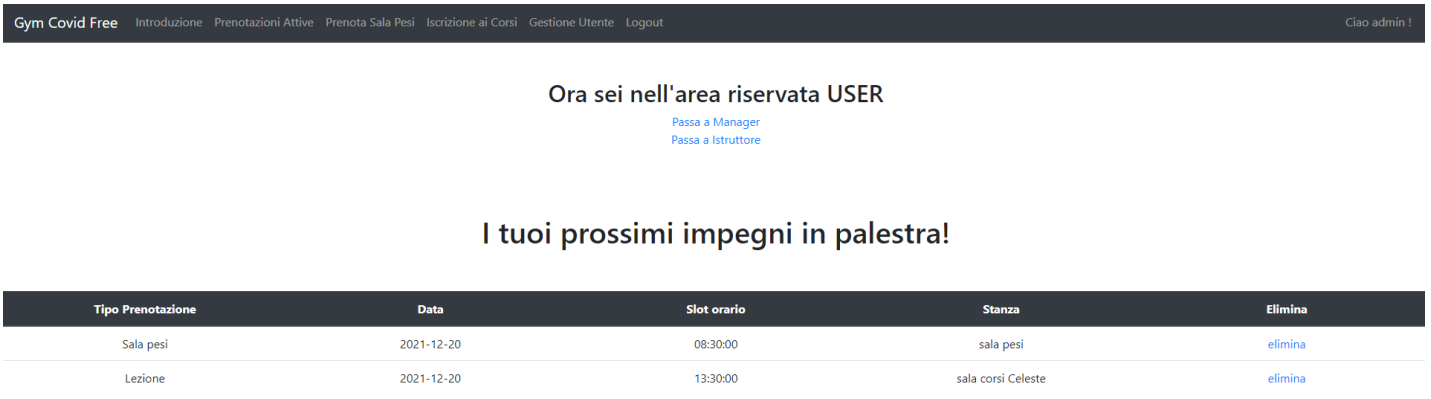

Qui l'user potrà registrare un nuovo account,se dovesse inserire un email già associata o non dovesse compilare un campo un avviso lo informerà dell'errore.
Alla creazione dell'account l'user verrà reindirizzato alla pagina di login

# Login



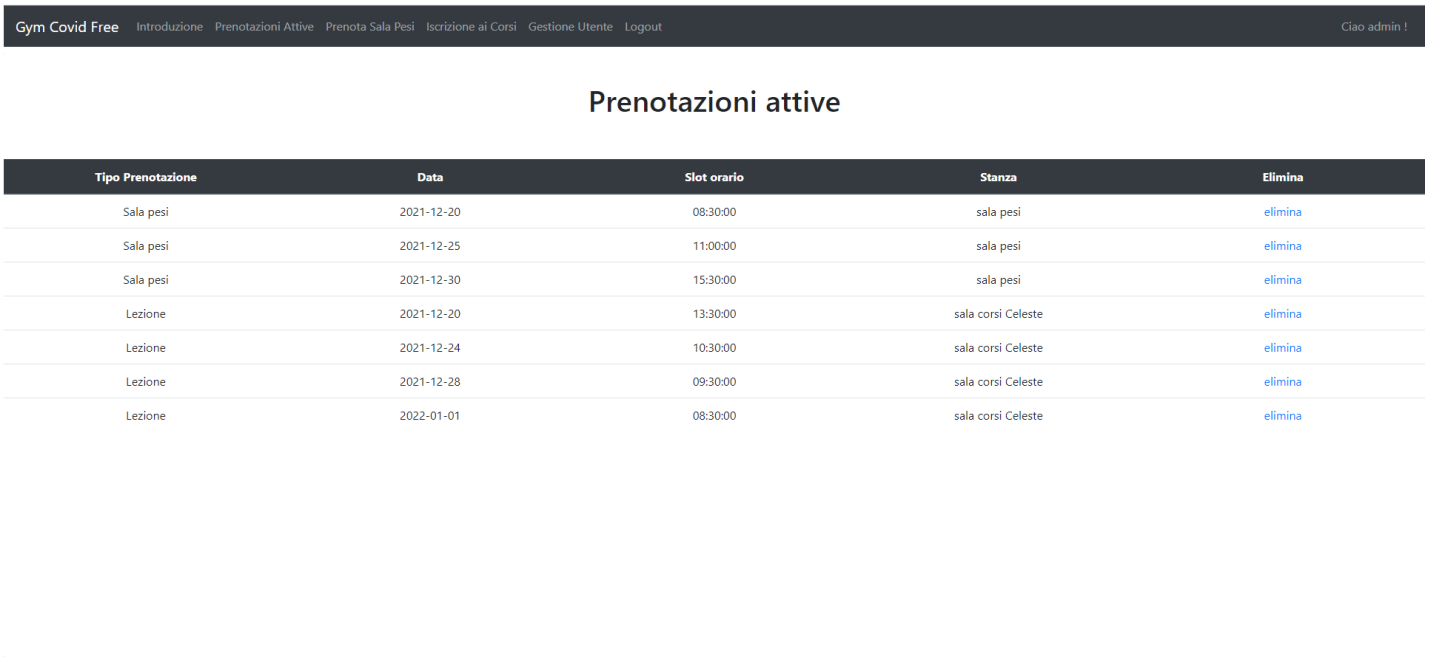

Qui l'user inserirà i dati dell'account,se non ci fosse corrispondenza tra i dati inseriti e quelli salvati nel db apparirà un avviso

# Introduzione User



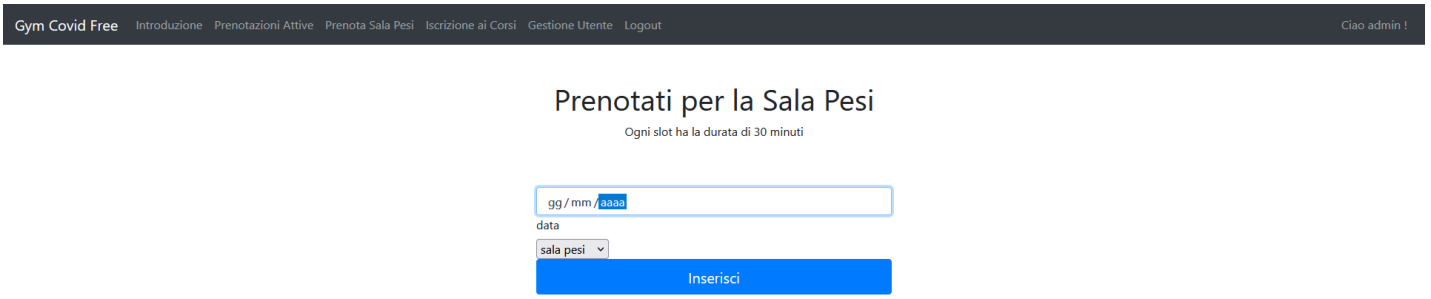

Da qui l'user potrà vedere i suoi prossimi impegni per 3 giorni ed eventualmente eliminarli

# Prenotazioni Attive



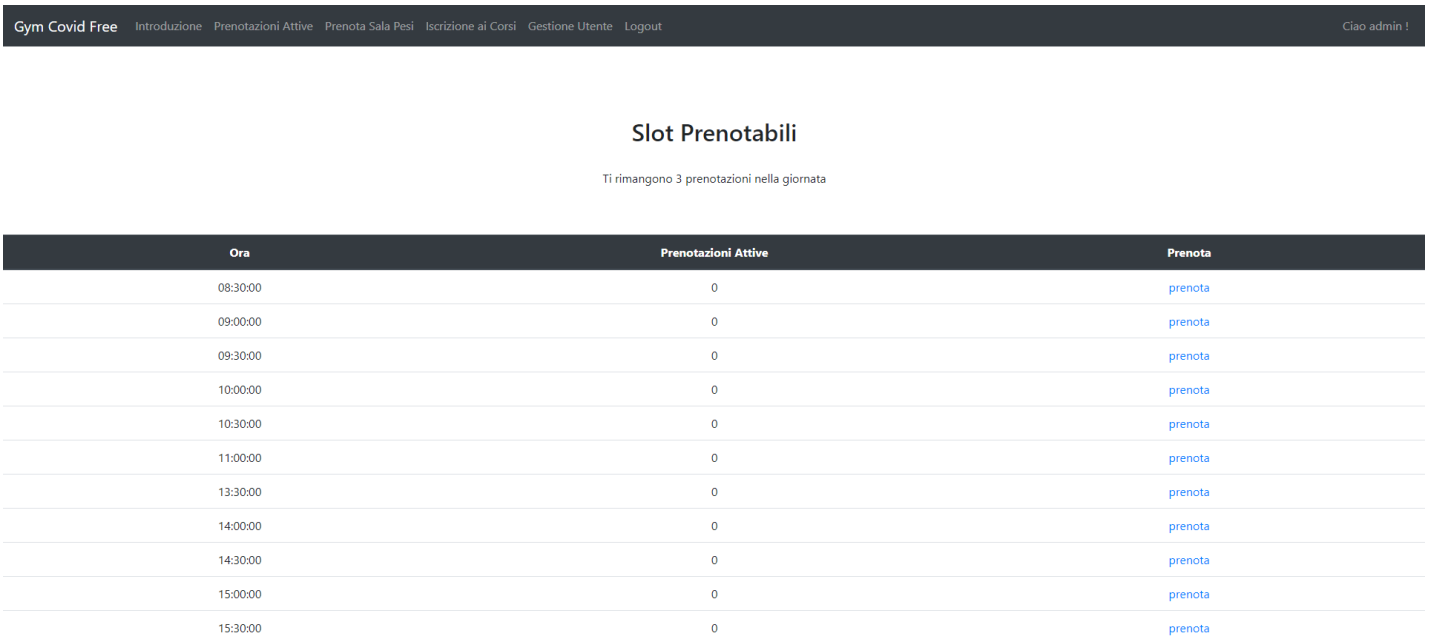

Da qui l'user può gestire tutte le sue prenotazioni da oggi in poi

# Prenota Sala Pesi



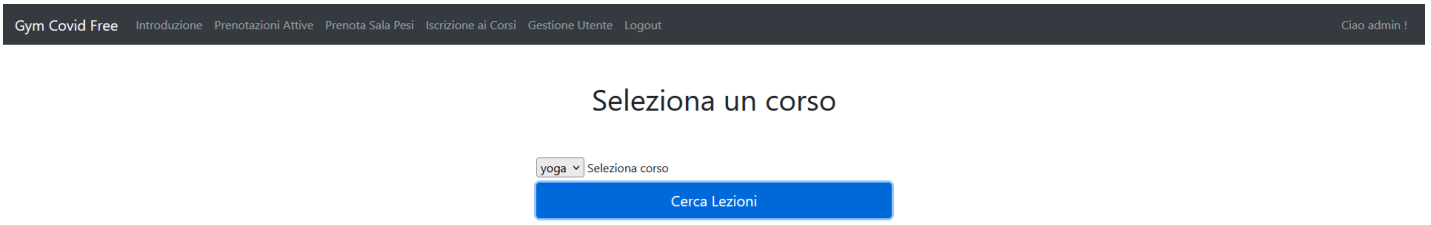

Da l'user può cercare degli slot prenotazione liberi in base alla giornata,alla sala e ai suoi slot rimanenti

# Slot Prenotabili



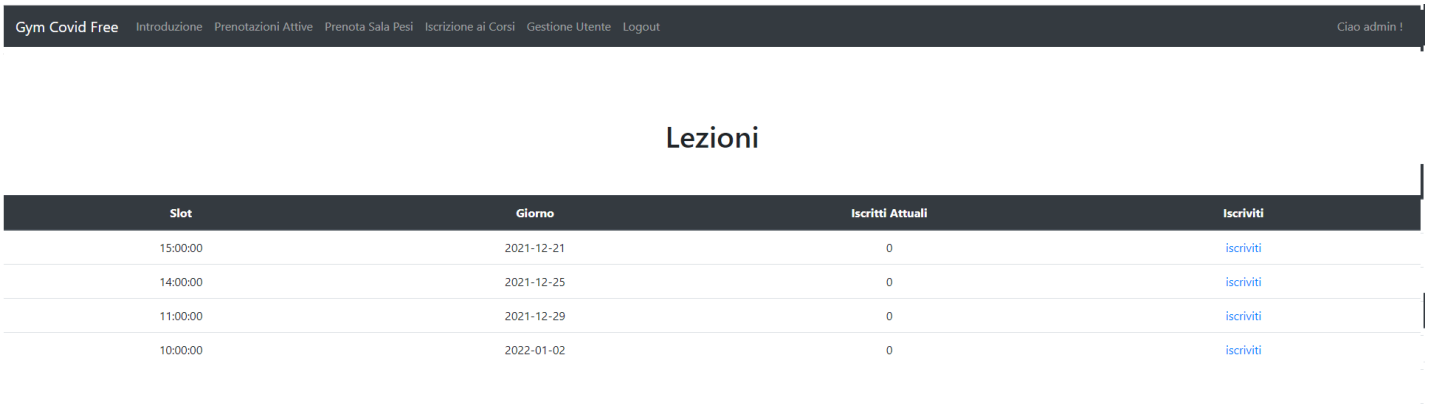

Dopo aver filtrato gli slot nella pagina precedente lo user potrà scegliere quale slot prenotare,vedendo al contempo quante persone si sono già iscritte in quello specifico slot e in quella specifica sala.
Grazie ad una sanitizzazione dei dati uno user non può vedere slot che ha già prenotato

# Iscrizione alle lezioni



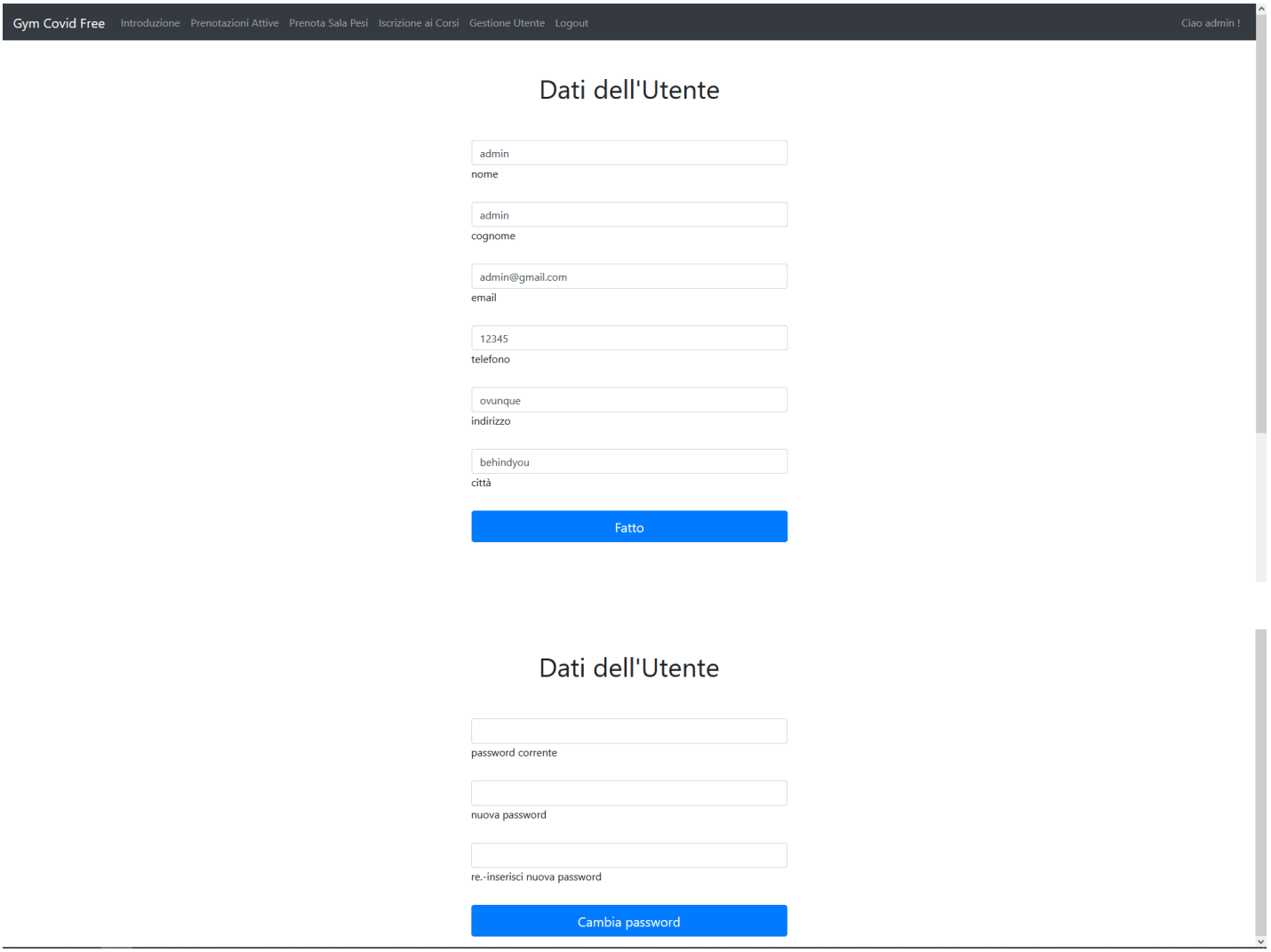

Da qui l'user può selezionare un corso e vedere le lezioni successive ad oggi alle quali non si è ancora iscritto

# Prenota Lezioni




Filtrati i corsi nella pagina precedente da qui l'utente vede le lezioni alle quali si può iscrivere e anche quanti partecipanti ci sono attualmente

# Gestione Utente

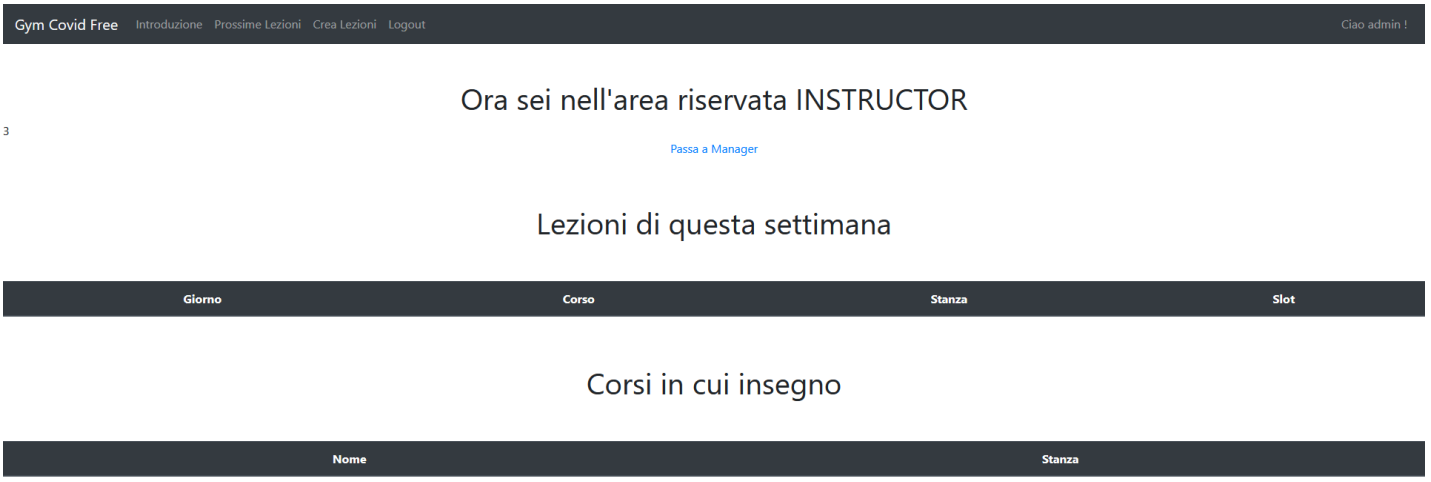






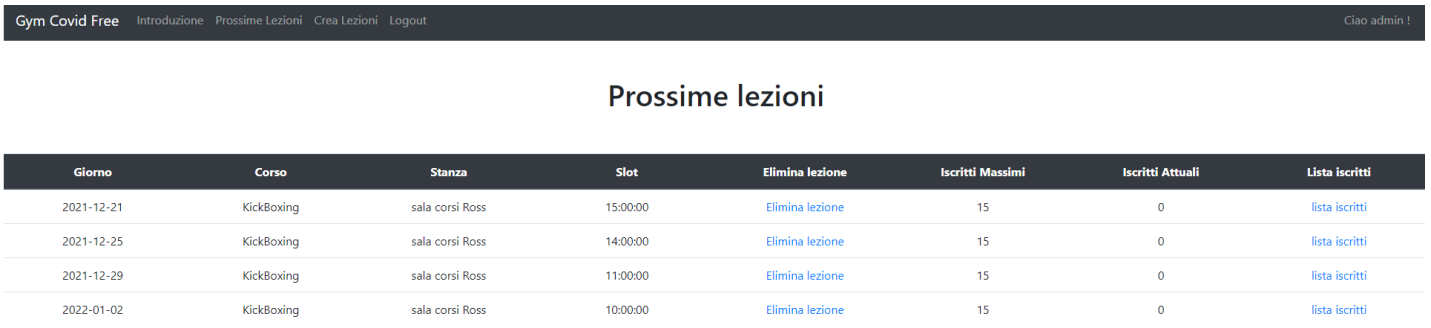
Da qui l’user può modificare i suoi dati(ad eccezione della mail)

# Introduzione Trainer



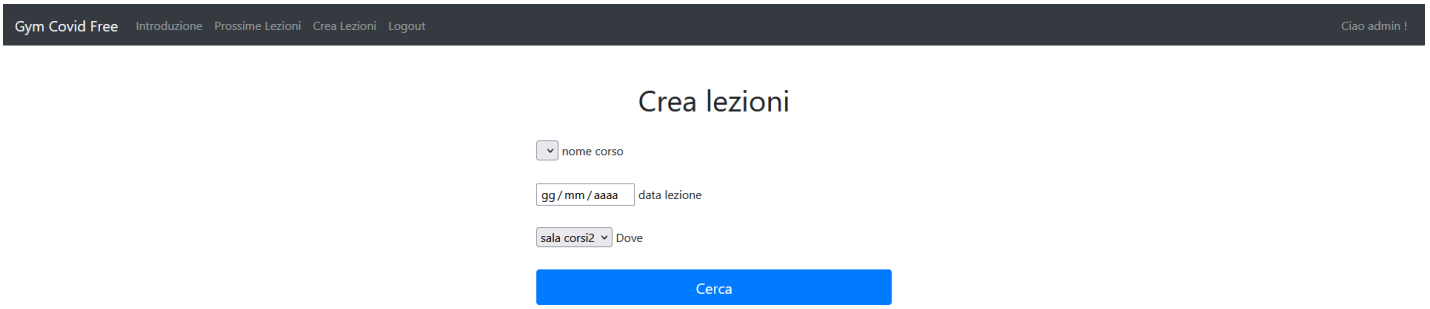

Da questa schermata l'istruttore ha una visuale sulle lezioni di questa settimana  in cui insegna con i relativi user e sui corsi a lui assegnati

# Prossime Lezioni

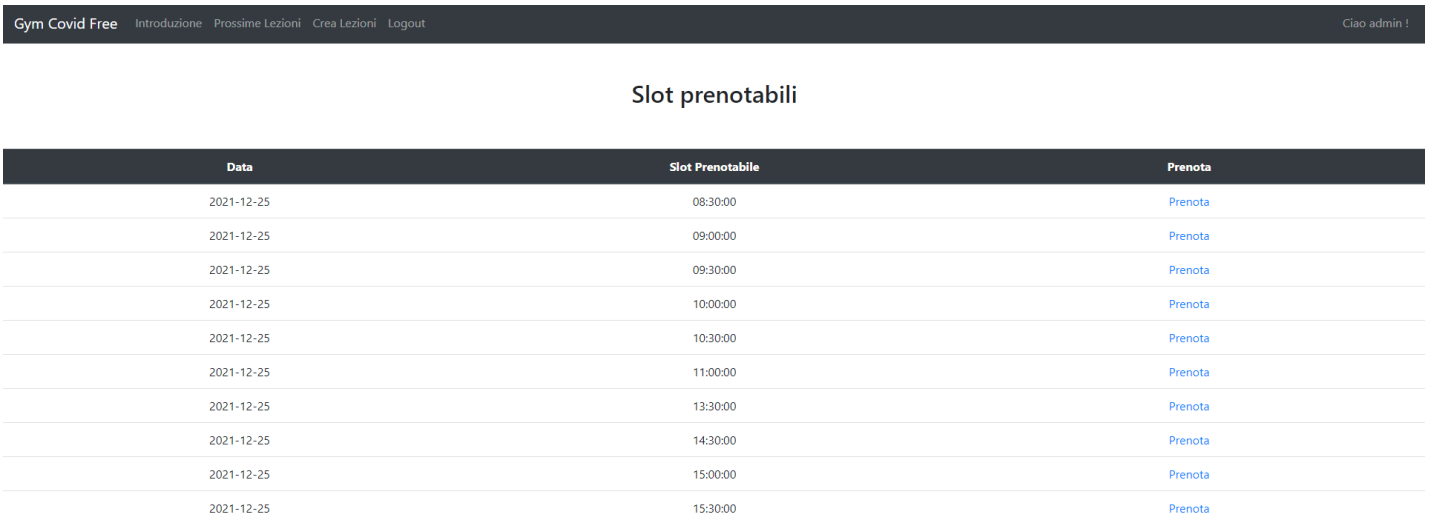



In questa schermata oltre ad avere una visione di tutte le prossime lezioni il trainer può anche gestirle(eliminarle/controllare gli iscritti etc)

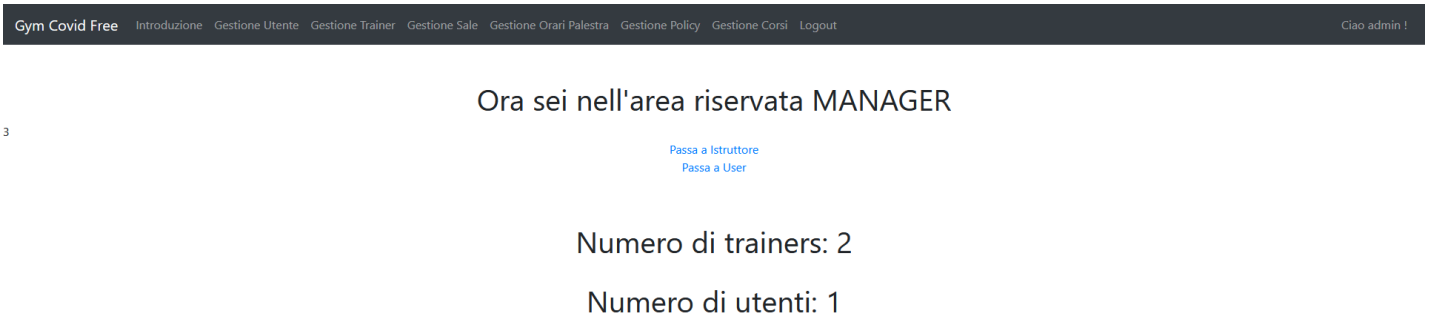
# Crea Lezioni




Da qui il Trainer seleziona un corso,una data e una sala,se la sala riesce ad ospitare il numero massimo di iscritti e degli slot sono liberi allora potrà creare una lezione nella successiva schermata

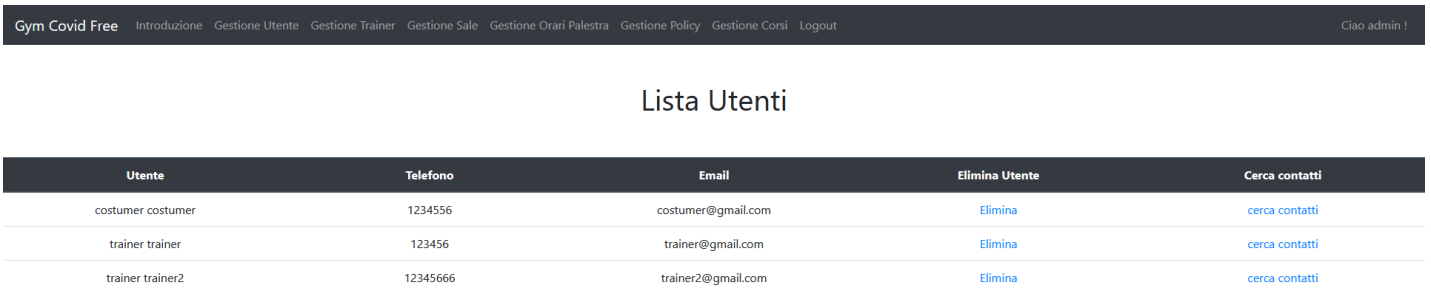


# Introduzione Manager

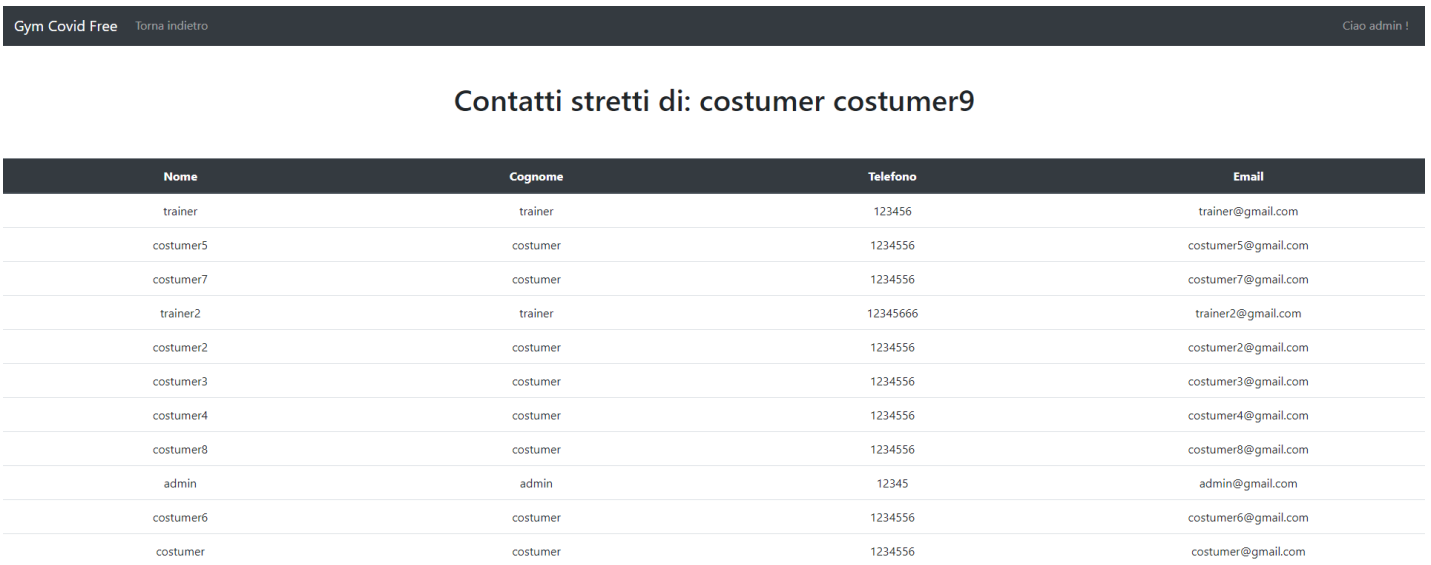



Da questa schermata il Manager ha una basica visione sul numero di trainers e clienti della palestra

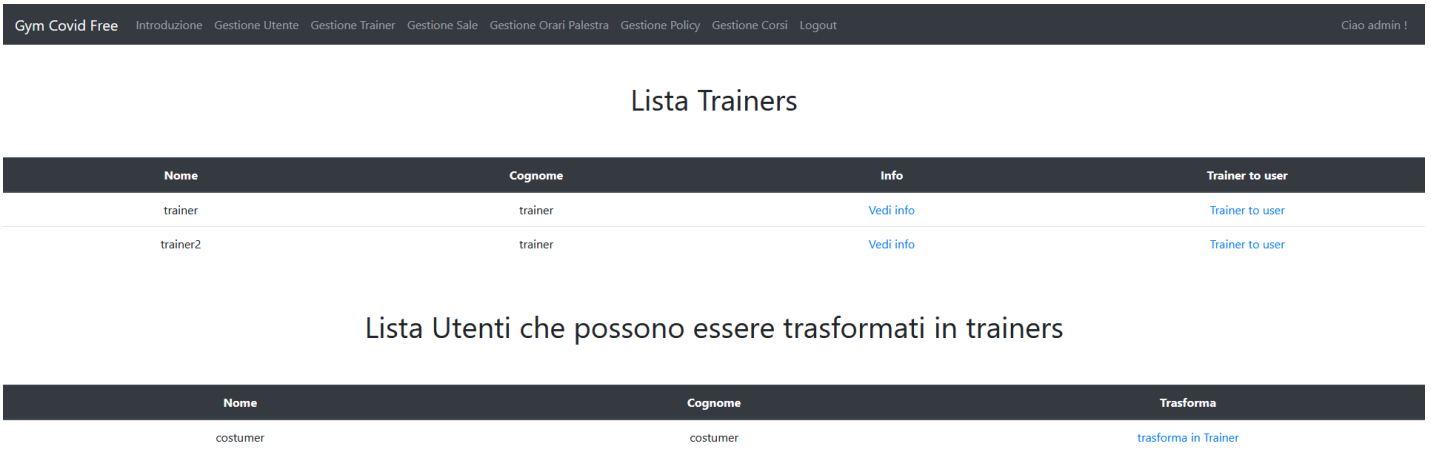
# Gestione Utente




Da qui il manager può visionare tutti gli attuali iscritti ,eliminarli ed effettuare il contact tracing nel caso qualcuno segnalasse la positività

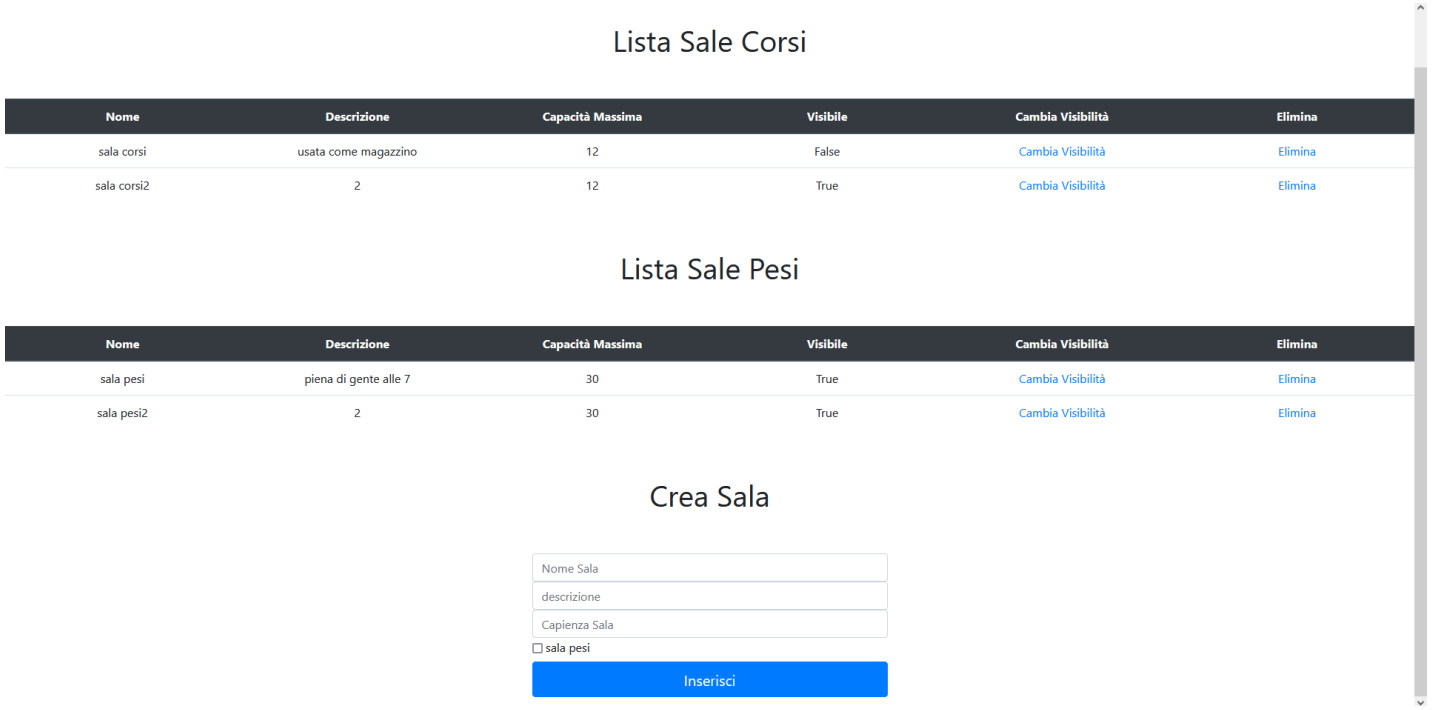



# Gestione Trainer




Questa schermata mostra i trainer attuali della palestra e da la possibilità di trasformarli in user e viceversa,per i trainer è possibile vedere anche quanti corsi tengono e quante lezioni hanno fatto

# Gestione Sale



Lista Sale Corsi

| Nome | Descrizione | Capacità Massima | Visibile | Cambia Visibilità | Elimina |
|---|---|---|---|---|---|
| sala corsi | usata come magazzino | 12 | False | Cambia Visibilità | Elimina |
| sala corsi2 | 2 | 12 | True | Cambia Visibilità | Elimina |

Lista Sale Pesi

| Nome | Descrizione | Capacità Massima | Visibile | Cambia Visibilità | Elimina |
|---|---|---|---|---|---|
| sala pesi | piena di gente alle 7 | 30 | True | Cambia Visibilità | Elimina |
| sala pesi2 | 2 | 30 | True | Cambia Visibilità | Elimina |

Crea Sala

Nome Sala

descrizione

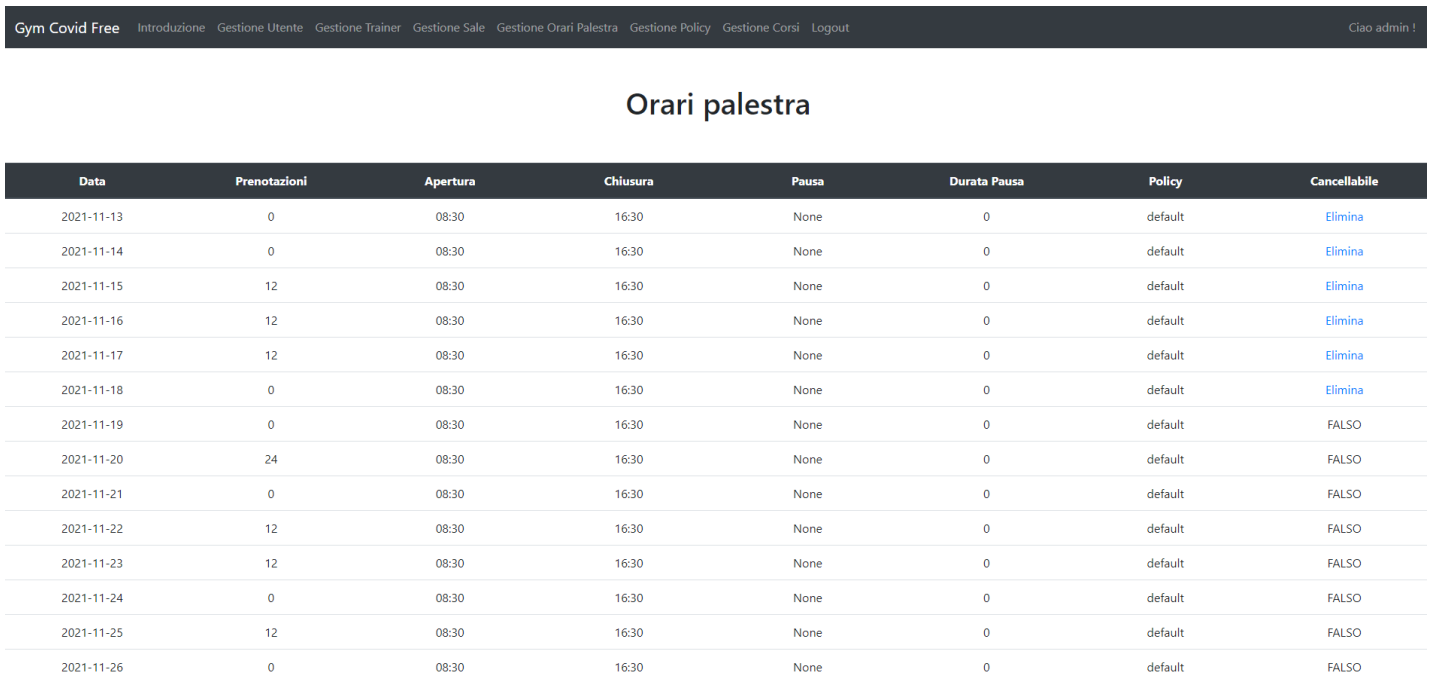
Capienza Sala

sala pesi

Inserisci

Qui il manager può creare sale,renderle invisibili ad users e trainer ed eliminarle

# Gestione Orari Palestra



Gym Covid Free Introduzione Gestione Utente Gestione Trainer Gestione Sale Gestione Orari Palestra Gestione Policy Gestione Corsi Logout Ciao admin !

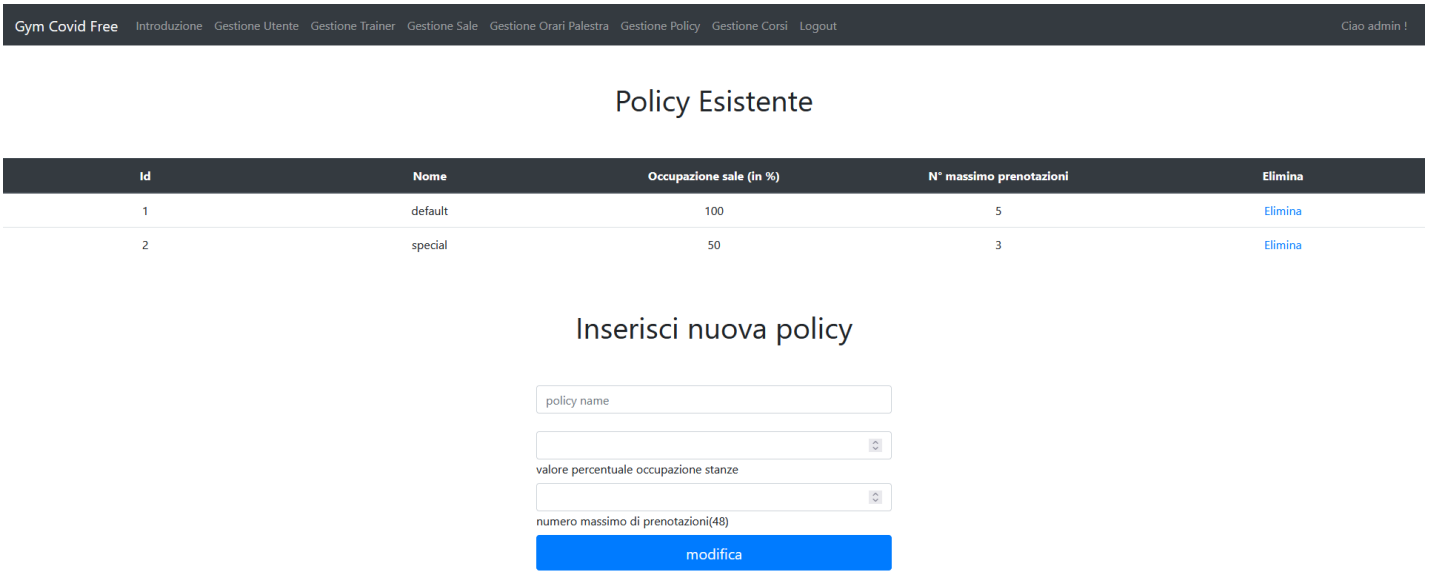
Orari palestra

| Data | Prenotazioni | Apertura | Chiusura | Pausa | Durata Pausa | Policy | Cancellabile |
|---|---|---|---|---|---|---|---|
| 2021-11-13 | 0 | 08:30 | 16:30 | None | 0 | default | Elimina |
| 2021-11-14 | 0 | 08:30 | 16:30 | None | 0 | default | Elimina |
| 2021-11-15 | 12 | 08:30 | 16:30 | None | 0 | default | Elimina |
| 2021-11-16 | 12 | 08:30 | 16:30 | None | 0 | default | Elimina |
| 2021-11-17 | 12 | 08:30 | 16:30 | None | 0 | default | Elimina |
| 2021-11-18 | 0 | 08:30 | 16:30 | None | 0 | default | Elimina |
| 2021-11-19 | 0 | 08:30 | 16:30 | None | 0 | default | FALSO |
| 2021-11-20 | 24 | 08:30 | 16:30 | None | 0 | default | FALSO |
| 2021-11-21 | 0 | 08:30 | 16:30 | None | 0 | default | FALSO |
| 2021-11-22 | 12 | 08:30 | 16:30 | None | 0 | default | FALSO |
| 2021-11-23 | 12 | 08:30 | 16:30 | None | 0 | default | FALSO |
| 2021-11-24 | 0 | 08:30 | 16:30 | None | 0 | default | FALSO |
| 2021-11-25 | 12 | 08:30 | 16:30 | None | 0 | default | FALSO |
| 2021-11-26 | 0 | 08:30 | 16:30 | None | 0 | default | FALSO |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2021-12-31 | 0 | 08:30 | 15:30 | 11:30 | 3 | special | Elimina |
| 2022-01-01 | 12 | 08:30 | 15:30 | 11:30 | 3 | special | FALSO |
| 2022-01-02 | 0 | 08:30 | 15:30 | 11:30 | 3 | special | FALSO |
| 2022-01-03 | 0 | 08:30 | 15:30 | 11:30 | 3 | special | FALSO |

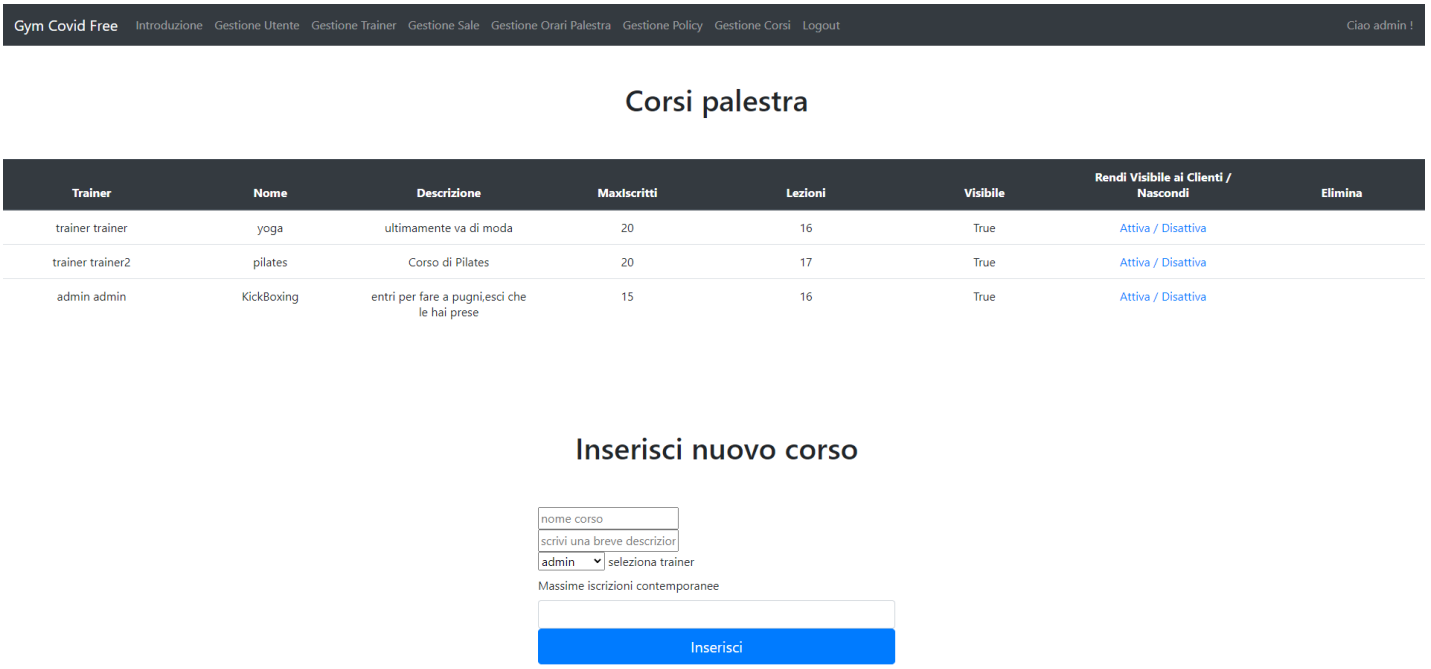

In questa schermata il manager può gestire e creare i giorni della settimana necessari ,in più riesce a vedere le prenotazioni della giornata

# Gestione Policy




Sezione per il managing delle policy da applicare ai giorni

# Gestione Corsi




In questa sezione è possibile creare corsi e renderli visibili o invisibili ai clienti,un corso è di default invisibile perchè non sono presenti lezioni al suo interno.

# Divisione compiti Progetto

Di seguito sono riportate le divisioni dei compiti per questo Progetto

| Sezione Progetto | Principale Sviluppatore | Sviluppatore di Supporto |
|---|---|---|
| Stesura Specifiche del tema | Furlan | Babato |
| Creazione Schema base di dati | Furlan | |
| Implementazione ORM | Furlan | |
| Implementazione Tecnologie Aggiuntive | Furlan | |
| Implementazione Sicurezza App | Furlan | |
| Implementazione Docker/Caricamento dati per Demo | Furlan | |
| Implementazione Politiche di | Furlan | |

| | | |
|---|---|---|
| integrità | | |
| Realizzazione Struttura Sito(HTML) | Babato | |
| Realizzazione Template(Jinja2) | Babato | Furlan |
| Realizzazione Blueprint | Babato | |
| Implementazione Endpoint | Babato | Furlan |
| Commenti al Codice | Babato/Furlan | |
| Stesura Documentazione | Furlan | Babato |