



CA' FOSCARI UNIVERSITY

BASI DATI PROJECT

Gym access during Covid-19 pandemic.

David Ambros, Florian Sabani, Ivan Spinello

supervised by
Sr. Sabani FLORIAN

December 13, 2021

Contents

| | | |
|-----------|---|-----------|
| 1 | Introduzione | 2 |
| 2 | Architettura | 2 |
| 3 | Installazione | 3 |
| 3.1 | Docker | 3 |
| 3.2 | Angular | 3 |
| 4 | Avvio | 3 |
| 4.1 | Avvio PostgreSQL | 3 |
| 4.2 | Avvio FlaskApp | 3 |
| 4.3 | Avvio AngularApp | 4 |
| 5 | Progetto | 4 |
| 6 | Progettazione del DB | 5 |
| 6.1 | Diagramma ad Oggetti | 7 |
| 6.2 | Attributi | 7 |
| 6.3 | Diagramma Relazionale | 8 |
| 7 | Definizione delle entita' nel DB | 8 |
| 7.1 | User | 8 |
| 7.2 | Subscriptions | 8 |
| 7.3 | Policies | 9 |
| 7.4 | Courses | 9 |
| 7.5 | Accesses | 9 |
| 7.6 | Reservations | 9 |
| 7.7 | Lessons | 9 |
| 7.8 | Slots | 9 |
| 7.9 | WeightRoomReservations | 9 |
| 7.10 | LessonReservation | 9 |
| 8 | Scelte progettuali | 10 |
| 8.1 | UniqueConstraint | 10 |
| 8.2 | CheckConstraint | 10 |
| 8.3 | Triggers | 10 |
| 8.4 | Transactions | 12 |
| 8.5 | Viste | 13 |
| 8.6 | Indici | 14 |
| 9 | Ruoli e Permessi | 14 |
| 10 | SQL Injections | 15 |
| 11 | Fun code blocks | 15 |
| 11.1 | Register | 15 |
| 11.2 | Login | 16 |
| 11.2.1 | Role Check | 16 |
| 11.3 | Response | 16 |
| 12 | Swagger e Documentazione | 17 |
| 13 | Image Upload & Download | 20 |

| | | |
|-----------|-----------------------------------|-----------|
| 14 | CI/CD | 21 |
| 14.1 | GitHub Actions | 21 |
| 14.2 | GitHub Secrets | 21 |
| 14.3 | DockerHub Registry | 22 |
| 14.4 | WebHook Server | 22 |
| 14.5 | Shell Redeploy Script | 22 |
| 15 | Componenti Angular | 22 |
| 16 | Commits | 28 |
| 17 | Trello & Agile Working | 29 |

Gym Project

Ca' Foscari University

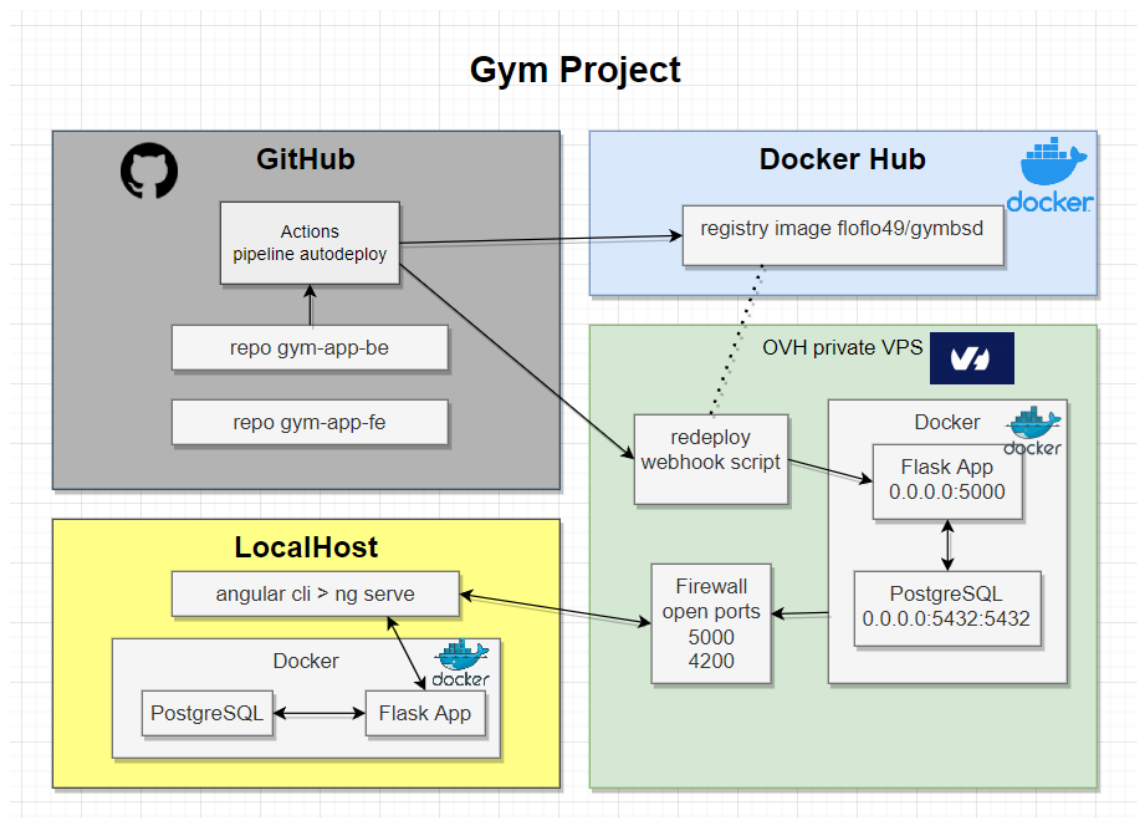
December 13, 2021

1 Introduzione

Per la realizzazione di questo progetto, l'applicazione è stata sviluppata con PyCharm in Flask, utilizzando SQLAlchemy Core (ORM) per l'interfacciamento con il DBMS sottostante (PostgreSQL). Le pagine per la web app sono state sviluppate con WebStorm in Angular 13.0.3. La grafica di base è stata realizzata prendendo spunto dai templates di Bulma (<https://bulma.io>). Le pagine html non vengono quindi renderizzate da Flask ma avviene una comunicazione tra back-end (flask) e front-end (angular) tramite il protocollo http in formato Json/REST. Per quanto riguarda il database e' stato utilizzato DataGrip come ide per la visione e simulazione delle query.

2 Architettura

Per poter comprendere a pieno come installare ed avviare l'applicazione e' necessario avere un'idea sull'architettura generale del progetto. E' chiaramente possibile eseguire l'intero progetto localmente, ma per poter lavorare in gruppo in modo piu' efficiente abbiamo deciso di avere sempre una versione aggiornata hostata in un server centralizzato accessibile a tutti. Inoltre onde evitare sprechi di tempo per i numerosi rilasci che questa scelta di lavoro ha richiesto, abbiamo deciso di sviluppare una pipeline di CI/CD per l'autodeploy del servizio back-end Flask.



L'architettura utilizzata per lo sviluppo di questo progetto si divide in quattro blocchi principali :

1. **GitHub**: repository per il versionamento del codice sia del backend che del frontend, utilizzato inoltre per la definizione delle Actions per le pipeline di CI/CD.
2. **DockerHub**: registry contenente le immagini docker derivate dalla docker-image del progetto Flask.
3. **OVH VPS**: Server privato virtuale che si presta per hostare DB e server app.
4. **LocalHost**: Macchina locale di ogni sviluppatore utilizzata per lo sviluppo e il testing locale prima di ogni commit.

3 Installazione

3.1 Docker

Per il test locale del progetto backend sara' necessaria l'installazione di :

- [docker](#)
- [docker-compose](#)

E - **basta!** no python no postgres no virtual enviroment...

3.2 Angular

Per il front-end e' necessario angular (a meno che non si intenda dockerizzare pure il FE) :

- [angular cli](#)

A meno che non si intenda usare la versione deployata nel server OVH del be, in tal caso sara' sufficiente avviare il fe settando il puntamento del be non localhost ma verso il server di riferimento.

4 Avvio

Per l'avvio locale dell'applicativo sara necessario seguire i seguenti step :

1. `docker-compose up -d`
2. `docker build -t flask:0.1 .`
3. `docker run -dit -p 5000:5000 flask:0.1`

4.1 Avvio PostgreSQL

Il primo comando si presta a lanciare il seguente file in modalita' detach

```
1 version: '3.5'
2 services:
3   database:
4     container_name: postgres
5     image: postgres:latest
6     env_file: database.conf
7     ports:
8       - 5432:5432
9     volumes:
```

Listing 1: docker-compose.yml

Il docker-compose come si vede a riga 6 necessita di un file database.conf situato nella stessa cartella costruito in questo modo :

```
1 POSTGRES_USER=<database user name>
2 POSTGRES_PASSWORD=<database pwd>
3 POSTGRES_HOST=localhost
4 POSTGRES_PORT=5432
5 POSTGRES_DB=<database name>
```

Listing 2: database.conf

L'immagine di postgres viene avviata preimpostando le sue variabili d'ambiente come definito dal database.conf file. A questo punto avremo localmente un container postgresQL avente un database in ascolto nella porta 5432, accessibile con l'ide usando username e pwd impostati nel database.conf file.

4.2 Avvio FlaskApp

A questo punto sara' necessario costruire l'immagine del nostro server flask col secondo comando, che non fa altro che eseguire il Dockerfile presente nella stessa cartella :

```
1 FROM python:3.8
2 ENV FLASK_APP=src/example/app.py
3 ENV FLASK_RUN_HOST=0.0.0.0
4
5 COPY requirements.txt requirements.txt
6 RUN pip install -r requirements.txt
7
8 ENV POSTGRES_USER=test
9 ENV POSTGRES_PASSWORD=password
10 ENV POSTGRES_HOST=localhost
11 ENV POSTGRES_PORT=5432
12 ENV POSTGRES_DB=example
13
14 EXPOSE 5000
15 COPY . .
16 CMD ["flask", "run"]
```

Listing 3: Dockerfile

Creata l'immagine flask potremo avviarla col terzo ed ultimo comando, specificando la porta (in questo caso 5000). A questo punto le rotte del nostro servizio si potranno gia' testare e chiamare con un programma come post-man.

4.3 Avvio AngularApp

Ma per completare il giro dovremmo anche avviare il front-end sempre localmente con il seguente comando :

1. `ng serve`

Possiamo ora navigare in `localhost:4200` per vedere la pagina home del progetto renderizzata sul nostro browser.

Link dei repo GitHub :

1. back-end [gym-app-be](#)
2. front-end [gym-app-fe](#)

Link dei repo DockerHub :

1. flask-app [gymbds](#)

5 Progetto

Il progetto si basa sulla realizzazione di un'applicazione web per la gestione di una palestra. Il nome della palestra è "Zyzz Gym". L'idea di base è realizzare una web application dove si ha la possibilità, come clienti, di poter prenotare un posto in una sala pesi oppure prenotare un posto per una lezione di uno specifico corso in una sala corsi. Rispettivamente, nel primo caso i clienti potranno usufruire degli strumenti e/o attrezzi che la palestra mette loro a disposizione (come bilancieri, manubri e macchinari). Nel secondo caso il cliente potrà partecipare a una lezione tenuta da un personal trainer (come lezioni di yoga, crossfit, zumba, box, etc.). A causa della situazione pandemica attuale, la palestra adotterà delle politiche di sicurezza come ad esempio la capienza ridotta delle stanze così da poter regolare il flusso di persone che vi potrà accedere, il numero di prenotazioni massime per giorno che un cliente può effettuare, orari di apertura e chiusura dettati dalle disposizioni del Governo e comportamenti idonei per un accesso sicuro alla palestra.

L'area comune parte dalla home page dove chiunque può accedervi, anche se non autenticato. Da qui si può accedere a diverse sezioni del sito:

- Register (per registrarsi come nuovi clienti)
- Login (per autenticarsi)
- Area personale (per vedere e modificare i propri dati)
- Calendario Corsi, Lezioni e Sale (per potercisi iscrivere)

In seguito al Login la sessione sarà valida per 30 minuti, dopodiché sarà necessario rifare il login un'altra volta.

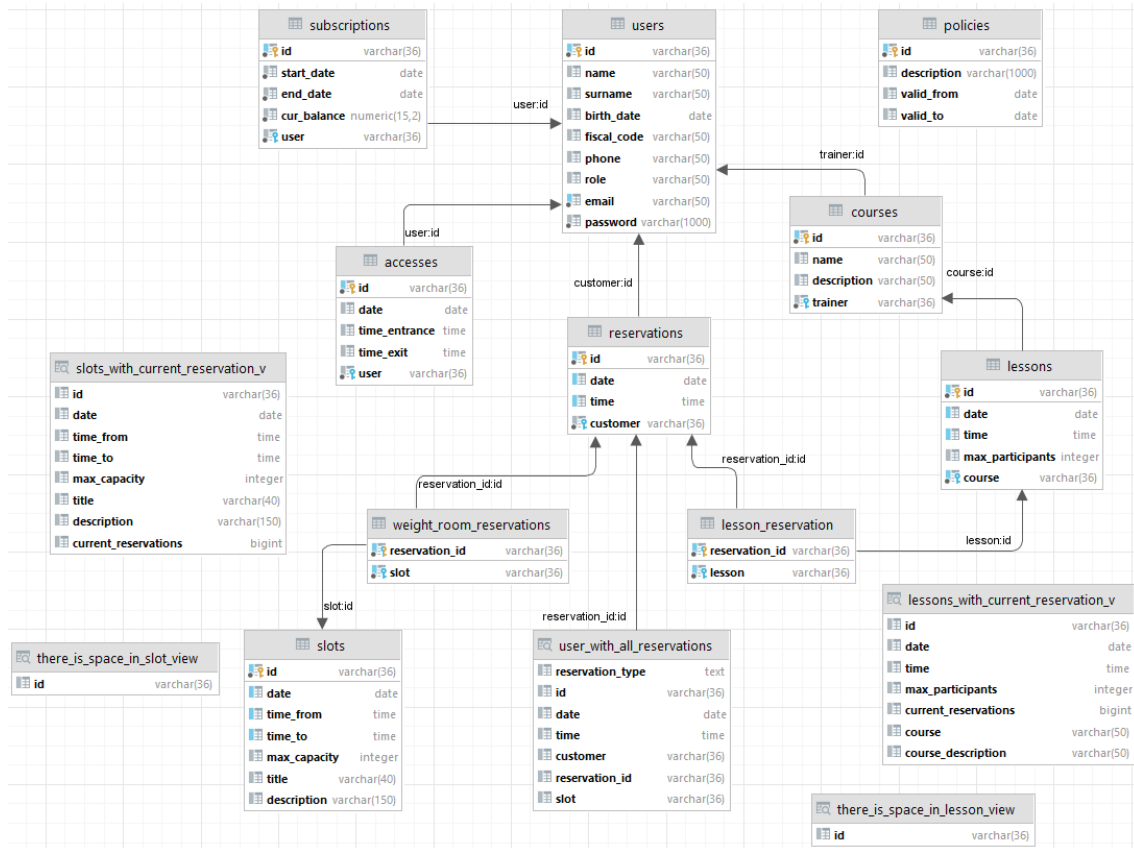
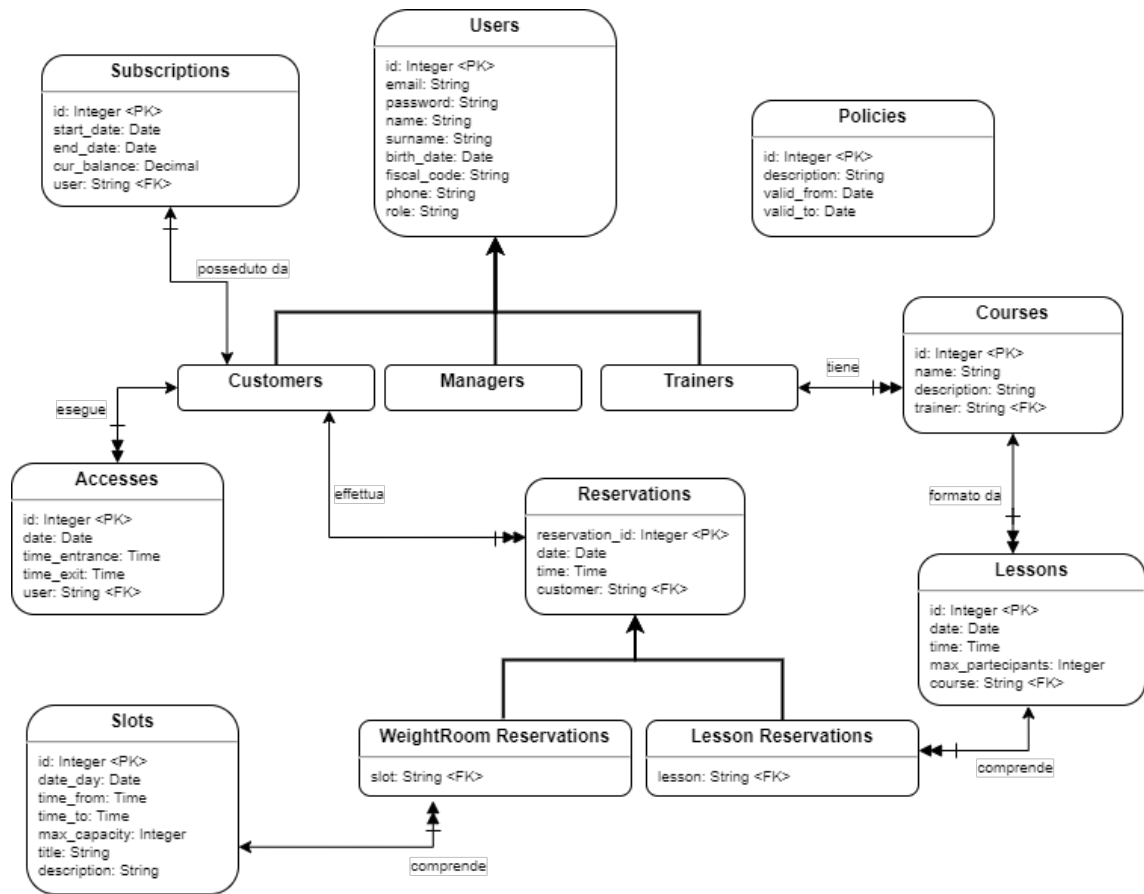
Clienti e staff della palestra avranno schermate differenti in base alle loro autorizzazioni erogate tramite i ruoli, in particolare:

- il manager potrà gestire l'inserimento di slots quindi i giorni e gli orari di apertura della sala pesi, modificare le policy, gestire le sottoscrizioni degli utenti etc.
- i personal trainer potranno inserire nuove lezioni e nuovi corsi.
- i customer possono prenotarsi a lezioni/corsi o slot per sale pesi.
- l'utente admin ha totale accesso a tutte le operazioni.
- l'utente machine ha potere di insert & update nella tabella dei accessi di un qualsiasi utente.

L'ultimo ruolo verrà assegnato esclusivamente alle macchinette che si occuperanno, tramite un apposito script, di inviare richieste di nuovo accesso o uscita dalla palestra di un utente.

6 Progettazione del DB

È necessario progettare una base di dati adeguata per fornire un sistema di gestione dei dati della palestra agli admin, trainers e clienti. Per implementare una base di dati, per prima cosa si analizza la realtà che si vuole andare a descrivere, creandovi un modello. Si identificano di seguito e relazionano le entità che fanno parte di questa realtà.



6.1 Diagramma ad Oggetti

Users: questa entità rappresenta tutte le persone che usfruiranno della palestra. Vengono salvate le informazioni personali quali nome e cognome, numero di telefono, codice fiscale e data di nascita; l'email e la password utili per il login nel sito, e il ruolo che descrive la tipologia di utente, descritte sopra.

Reservations: questa entità rappresenta tutti i tipi di prenotazioni conseguibili dai clienti della palestra. Ogni prenotazione ha una propria data e ora, e può essere di due tipi:

- **WeightRoom Reservations:** prenotazioni relative alla sala pesi, con il proprio relativo slot orario.
- **Lessons Reservations:** prenotazioni relative ad una lezione di un corso.

Slots: rappresenta una fascia oraria per l'accesso alla sala pesi. Ogni slot ha un'orario di inizio e di fine, e una capacità massima dato che un numero limitato di clienti potranno usufruire della sala pesi durante una fascia oraria. Ogni slot ha anche una descrizione aggiuntiva.

Subscriptions: rappresenta tutti gli abbonamenti dei clienti, con data di inizio e fine.

Accesses: rappresenta uno storico di tutti gli accessi in palestra da parte degli utenti. Ogni accesso ha relativa data, ora entrata e ora uscita.

Courses: rappresenta i tipi di corsi tenuti dai trainer della palestra, con relativa descrizione. **Lessons:** rappresenta uno storico di tutte le lezioni dei corsi tenutesi, con relativa data e ora, numero di partecipanti massimi, e relativo corso.

Policies: rappresenta una policy della palestra, con relativa descrizione e tempo di validità.

Definite le entità, si possono delineare le loro relazioni.

- **Customer (esegue) Accesses:** un cliente accederà alla palestra più volte, anche durante la stessa giornata. Relazione uno a molti.
- **Subscriptions (posseduto da) Customers:** ogni cliente avrà uno storico di tutte le iscrizioni effettuate. Relazione uno a molti.
- **Trainers (tiene) Courses:** un trainer terrà uno o più corsi specifici nella palestra. Relazione uno a molti.
- **Customers (effettua) Reservations:** ogni cliente avrà uno storico di tutte le prenotazioni effettuate. Relazione uno a molti.

- **Courses (svolto in) Lessons:** ogni corso avrà uno storico di tutte le lezioni svoltesi. Relazione uno a molti.
- **WeightRoomReservations (comprende) Slots:** storico delle prenotazioni della sala pesi con relativo slot orario. Relazione uno a molti.
- **LessonReservations (comprende) Lessons:** storico delle prenotazioni delle lezioni del relativo corso. Relazione uno a molti.

6.2 Attributi

- **Users**
 - id** id univoco per il cliente
 - name**
 - surname**
 - birth date**
 - fiscal code**
 - phone**
 - role** ruolo specifico dell'utente, in base a questo attributo avrà accesso a diversi livelli di funzionalità.
 - email** email necessaria per il login nel sito.
 - password** password necessaria per il login nel sito.
- **Reservations**
 - id** id univoco per la prenotazione generica
 - date** giorno della prenotazione
 - time** orario inizio prenotazione
 - customer** id univoco del cliente della prenotazione
- **Subscriptions**
 - id** id univoco dell'iscrizione di un cliente
 - start date** giorno di inizio validità dell'iscrizione
 - end date** giorno di fine validità dell'iscrizione
 - user** id univoco del cliente dell'iscrizione
- **Accesses**
 - id** id univoco dell'accesso di un cliente
 - date** giorno dell'accesso
 - time entrance** orario di entrata alla palestra
 - time exit** orario di uscita dalla palestra
 - user** id univoco dell'utente entrato in palestra
- **Courses**
 - id** id univoco del corso

name
description breve descrizione scritta del corso
trainer id univoco del trainer che terrà il corso

- Lessons
id id univoco per la singola lezione
data giorno in cui si terrà la lezione
time orario di inizio lezione
max participants numero massimi di possibili prenotazioni per la lezione
course id univoco del tipo di corso svolto
- Slots
id id univoco dello slot orario
date giorno dello slot orario
time from orario inizio slot orario
time to orario fine slot orario
max capacity numero di prenotazioni massime per lo slot orario
title breve titolo dello slot orario
description breve descrizione scritta dello slot orario
- Policies
id id univoco della policy
description breve descrizione scritta della policy
valid from inizio validità della policy
valid to fine validità della policy

6.3 Diagramma Relazionale

La progettazione logica della base di dati consiste nella traduzione dello schema ad oggetti in uno schema concettuale che rispecchia il modello dei dati, nel nostro caso relazionale. Nel modello relazionale le entità comprendono vincoli di chiave primaria (PK), chiave esterna (FK) e vincoli NOT NULL.

Nel passaggio dal diagramma ad oggetti a relazionale, le gerarchie di entità devono essere ripartizionate. Nella nostra implementazione ci sono due gerarchie: sulla tabella Users e Reservations. Per la gerarchia Users, il partizionamento ovvio è quello della tabella unica, dato che nelle sottoclassi non ci sono attributi di differenza tra i diversi tipi di utenti (il ruolo è determinato da un attributo).

Per la gerarchia Reservations, è stato scelto un partizionamento verticale, quindi con 3 tabelle, una per la prenotazione generica, e 2 per le prenotazioni specifiche (sala pesi e lezione). Questa scelta implementativa centralizza tutte le prenotazioni in unico punto.

Il partizionamento verticale è utile nel caso in cui si voglia determinare per un qualsiasi utente quante prenotazioni sono assegnate al suo ID in un determinato lasso di tempo, senza il bisogno di eseguire una join della tabella User e Reservations. Difatti l'ID dell'utente è salvato come chiave esterna nella tabella della relativa Reservation.

7 Definizione delle entità nel DB

Sono necessarie delle definizioni precise in base alla realtà che si vuole descrivere e per riuscire a mantenere una situazione di integrità dei dati all'interno della base relazionale. Per vedere la loro definizione e' sufficiente andare nella sezione del repository [contenente i modelli](#). Onde evitare un copy/paste papale, verranno riportate a seguito solo alcune configurazioni in pseudo-codice:

7.1 User

```
1 class Users(db.Model):
2     __tablename__ = "users"
3     __table_args__ = (UC("email"))
4
5     id = db.Column(ID_TYPE, pk=True)
6     name = db.Column(db.String(50), ...)
7     surname = db.Column(db.String(50))
8     birth_date = db.Column(db.Date, ...)
9     fiscal_code = db.Column(db.String)
10    phone = db.Column(db.String(50))
11    role = db.Column(db.String(50))
12    email = db.Column(db.String(50))
13    password = db.Column(db.String(100))
```

Listing 4: Users db table

7.2 Subscriptions

```
1 class Subscriptions(db.Model):
2     __table_name__ = "Subscriptions"
3     __table_args__ = (
4         CK('end_date > start_date'))
5
6     id = db.Column("id", ID_TYPE, ...)
7     start_date = db.Column(db.Date, ...)
8     end_date = db.Column(db.Date, ...)
9     cur_balance = db.Column(db.Numeric(15, 2), CK("cur_balance > 0"))
10    user = db.Column(db.FK(Users.id))
```

Listing 5: Subscriptions db table

7.3 Policies

```
1 class Policies(db.Model):
2     __table_name__ = "Policies"
3     __table_args__ = (
4         CK('valid_to > valid_from'))
5
6     id = db.Column("id", ID_TYPE, ...)
7     description = db.Column(db.String)
8     valid_from = db.Column(db.Date)
9     valid_to = db.Column(db.Date)
```

Listing 6: Policies db table

7.4 Courses

```
1 class Courses(db.Model):
2     __table_name__ = "Courses"
3     id = db.Column("id", ID_TYPE...)
4     name = db.Column("name", db.String)
5     description = db.Column(db.String)
6     trainer = db.Column(db.FK(Users.id))
```

Listing 7: Courses db table

7.5 Accesses

```
1 class Accesses(db.Model):
2     __table_name__ = "Accesses"
3     __table_args__ = (
4         CK('time_exit > time_entrance')
5     )
6     id = db.Column("id", ID_TYPE)
7     date = db.Column("date", db.Date)
8     time_entrance = db.Column(db.Time)
9     time_exit = db.Column(db.Time)
10    user = db.Column(db.FK(Users.id))
```

Listing 8: Accesses db table

7.6 Reservations

```
1 class Reservations(db.Model):
2     __table_name__ = "Reservations"
3     __table_args__ = (
4         UC("date", "time")
5     )
6     id = db.Column("id", ID_TYPE)
7     date = db.Column("date", db.Date)
8     time = db.Column("time", db.Time)
9     customer = db.Column(db.FK(Users.id))
```

Listing 9: Reservations db table

7.7 Lessons

```
1 class Lessons(db.Model):
2     __table_name__ = "Lessons"
3     __table_args__ = {
4         UC("course", "date", "time")
5     }
```

```
6     id = db.Column("id", ID_TYPE)
7     date = db.Column("date", db.Date)
8     time = db.Column("time", db.Time)
9     max_participants = db.Column(db.Integer, CheckConstraint("
10    max_participants > 0"))
11    course = db.Column(db.FK(Courses.id))
```

Listing 10: Lessons db table

7.8 Slots

```
1 class Slots(db.Model):
2     __table_name__ = "Slots"
3     __table_args__ = (
4         UC("time_from", "time_to", "date")
5         CK('time_to > time_from')
6     )
7
8     id = db.Column("id", ID_TYPE)
9     date = db.Column(db.Date)
10    time_from = db.Column(db.Time)
11    time_to = db.Column(db.Time)
12    max_capacity = db.Column(db.Integer, CK("max_capacity > 0"),
13    default=20)
14    title = db.Column("title", db.String)
15    description = db.Column(db.String)
```

Listing 11: Slots db table

7.9 WeightRoomReservations

```
1 class WeightRoomReservations(db.Model):
2     :
3     __table_name__ = "
4     WeightRoomReservations"
5     __table_args__ = (UC("slot", "
6     reservation_id"))
7
8     reservation_id = db.Column(ID_TYPE, db.FK(Reservations.id))
9     slot = db.Column(db.FK(Slots.id))
```

Listing 12: WeightRoomReservations db table

7.10 LessonReservation

```
1 class LessonReservation(db.Model):
2     __table_name__ = "
3     LessonReservation"
4     __table_args__ = (UC("lesson", "
5     reservation_id"))
6
7     reservation_id = db.Column(ID_TYPE, db.FK(Reservations.id))
8     lesson = db.Column(ID_TYPE, db.FK(Lessons.id))
```

Listing 13: LessonReservation db table

8 Scelte progettuali

Sono necessarie delle scelte progettuali utili per poter delineare al meglio tutte le modalità per implementare nella maniera più efficiente e pulita tutto il codice. Per questo motivo abbiamo deciso di implementare dei vincoli, trigger, check su attributi, transazioni utili per le prenotazioni e ruoli nella base di dati per gestire le autorizzazioni degli utenti.

Vincoli :

- **NOT NULL** Vi sono righe di tabelle che sono accompagnate dal vincolo Not Null. Infatti quelle righe non ammettono che non venga inserito alcun valore.
- **PRIMARY KEY** Tutte le tabelle presentano un codice identificativo come primary key in grado di identificare univocamente ogni entry inserita nella base di dati.
- **FOREIGN KEY** Il vincolo di Foreign Key è facile da posizionare per poter ottenere relazioni tra più tabelle collegandole tra di loro.
- **UNIQUE** Il vincolo Unique ci torna utile per garantire che non vengano immessi valori duplicati in colonne specifiche che non fanno parte di una chiave primaria.

8.1 UniqueConstraint

Abbiamo usato **UniqueConstraint** su:

1. **Users** per evitare di registrare lo stesso utente con la stessa mail più volte.
2. **Reservations** per evitare di registrare due reservations nello stesso giorno/orario.
3. **Lessons** per evitare di registrare lo stesso corso nello stesso giorno nello stesso orario.
4. **Slots** per evitare di aggiungere più di uno slot nello stesso intervallo di tempo.
5. **WeightRoomReservations** per assicurarci di non puntare sullo stesso slot/reservation.
6. **LessonReservation** per assicurarci di non puntare sulla stessa lesson/reservation.

8.2 CheckConstraint

Abbiamo usato **CheckConstraint** su:

1. Subscriptions per assicurarci che la enddate sia futura rispetto alla startdate in fase di inserimento.
2. Subscriptions per assicurarci che il bilancio corrente sia maggiore di 0. (non impl)
3. Policies per assicurarci che la policy che si intende inserire sia valida.
4. Accesses per controllare che l'orario di uscita sia conseguente a quello d'ingresso.
5. Lessons per controllare che il num di partecipanti iniziali sia maggiore di 0.
6. Slots per controllare che in fase di inserimento l'orario di fine sia conseguente a quello d'inizio.

8.3 Triggers

E' stato deciso di usare 4 trigger, il più essenziale e immediato è:

1. Quando viene fatto un inserimento/aggiornamento nella tabella corsi, dobbiamo essere sicuri che il trainer, che punta alla tabella Users tramite foreign-key, abbia effettivamente ruolo = 'trainer', e che non sia un customer, manager o altro.

Di seguito troviamo la funzione chiamata dal trigger:

```
1 CREATE OR REPLACE FUNCTION "gym".
  is_trainer_fun() RETURNS
  trigger AS $$
2
3 DECLARE
4   trainer_row "gym".users%
  ROWTYPE= NULL;
5
6 BEGIN
7   SELECT * INTO trainer_row
8   FROM "gym".users u
9   WHERE u.id=NEW.trainer;
10
11   IF trainer_row.role == '
  trainer' THEN
12     RETURN NEW;
13   ELSE
14     RETURN NULL;
15   END IF;
16 END
17 $$ LANGUAGE plpgsql;
```

Listing 14: is_trainer_fun

Mentre questo è il trigger sulla tabella Coruses:

```

1 DROP TRIGGER IF EXISTS is_trainer
  ON "gym".courses;
2 CREATE TRIGGER is_trainer
3 BEFORE INSERT OR UPDATE ON "gym".
  courses
4 FOR EACH ROW
5 EXECUTE FUNCTION "gym".
  is_trainer_fun();

```

Listing 15: is_trainer_fun_trigger

- Il secondo trigger controlla che le prenotazioni per la sala pesi non siano al limite per un determinato slot, cioè che lo slot non sia pieno e abbia raggiunto la sua max_capacity

```

1 CREATE OR REPLACE FUNCTION "gym".
  is_slot_full_fun() RETURNS
  trigger AS $$
2
3 DECLARE
4   slot_row "gym".slots%ROWTYPE=
  NULL;
5   current_occupation integer;
6
7 BEGIN
8   SELECT * INTO slot_row
9   FROM "gym".slots slot
10  WHERE slot.id=NEW.slot;
11
12   SELECT COUNT(*) INTO
13   current_occupation
14   FROM "gym".
15   weight_room_reservations wr
16   WHERE wr.slot = NEW.slot;
17
18   IF current_occupation>=
19   slot_row.max_capacity THEN
20     DELETE FROM "gym".
21     reservations r
22     WHERE r.id = NEW.
23     reservation_id;
24     RETURN NULL;
25   ELSE
26     RETURN NEW;
27   END IF;
28 END;
29 $$ LANGUAGE plpgsql;

```

Listing 16: is_slot_full_fun

```

1 DROP TRIGGER IF EXISTS
  is_slot_full ON "gym".
  weight_room_reservations;
2 CREATE TRIGGER is_slot_full
3 BEFORE INSERT OR UPDATE ON "gym".
  weight_room_reservations
4 FOR EACH ROW
5 EXECUTE FUNCTION "gym".
  is_slot_full_fun();

```

Listing 17: is_slot_full_fun_trigger

- Il terzo trigger fa la stessa cosa analoga del secondo, ma riferito alle lezioni:

```

1 CREATE OR REPLACE FUNCTION "gym".
  is_lesson_full_fun() RETURNS
  trigger AS $$
2
3 DECLARE

```

```

4   lesson_row "gym".lessons%
  ROWTYPE= NULL;
5   current_occupation integer;
6
7 BEGIN
8   SELECT * INTO lesson_row
9   FROM "gym".lessons lesson
10  WHERE lesson.id=NEW.lesson
11  ;
12
13   SELECT COUNT(*) INTO
14   current_occupation
15   FROM "gym".
16   lesson_reservation lr
17   WHERE lr.lesson = NEW.
18   lesson;
19
20   IF current_occupation+1>
21   lesson_row.max_participants
22   THEN
23     DELETE FROM "gym".
24     reservations r
25     WHERE r.id = NEW.
26     reservation_id;
27     RETURN NULL;
28   ELSE
29     RETURN NEW;
30   END IF;
31 END;
32 $$ LANGUAGE plpgsql;

```

Listing 18: is_lesson_full_fun

```

1 DROP TRIGGER IF EXISTS
  is_lesson_full ON "gym".
  lesson_reservation;
2 CREATE TRIGGER is_lesson_full
3 BEFORE INSERT OR UPDATE ON "gym".
  lesson_reservation
4 FOR EACH ROW
5 EXECUTE FUNCTION "gym".
  is_lesson_full_fun();

```

Listing 19: is_lesson_full_trigger

- Il quarto trigger si occupa di guardare che quando viene fatto un accesso alla palestra, l'utente in questione abbia un abbonamento valido cioè start-date< NOW() < end-date:

```

1 CREATE OR REPLACE FUNCTION "gym".
  check_subscription()
2 RETURNS trigger AS $$
3
4 DECLARE
5   subscription "gym".
  subscriptions%ROWTYPE=NULL;
6
7 BEGIN
8   SELECT * INTO subscription
9   FROM "gym".subscription s
10  WHERE s.user = NEW.user;
11
12   IF subscription.start_date <
13   NOW() AND subscription.end_date
14   > NOW() THEN
15     RETURN NEW;
16   ELSE
17     RETURN NULL;
18   END IF;
19 END

```

```
18 $$ LANGUAGE plpgsql;
```

Listing 20: is_subscription_valid_fun

```
1 DROP TRIGGER IF EXISTS
  check_subscription ON "gym".
  accesses;
2 CREATE TRIGGER check_subscription
3 BEFORE INSERT OR UPDATE ON "gym".
  accesses
4 FOR EACH ROW
5 EXECUTE FUNCTION "gym".
  check_subscription();
```

Listing 21: is_subscription_valid_trigger

8.4 Transactions

In questo progetto abbiamo deciso di utilizzare le transazioni per gestire tutte le operazioni sui dati. Per implementare cio' abbiamo sfruttato una funzionalita' offerta dalla libreria di SQLAlchemy. Ciò ci permette di non lasciare uno stato inconsistente dei dati all'interno della base di dati e operare con i dati aggiornati alle reali ultime modifiche fatte su di essi. In ogni operazione viene eseguito un blocco contenuto in un'istruzione "try: except:" in cui andiamo ad effettuare l'operazione che ci interessa, seguita da un commit(), nel blocco "try"; mentre nel "except" andiamo ad eseguire un rollback() se l'operazione di commit() non va a buon fine. In questo modo l'eventuale aggiunta, aggiornamento o eliminazione di un elemento della sessione viene annullata e ripristinata la situazione precedente.

Un esempio di funzione in cui utilizziamo il meccanismo sopra descritto puo' essere il seguente :

```
1 ## -- security.py -- ##
2 def register_user(data):
3     database.add_instance( # <- usage
4         Users,
5         id=id_user,
6         email=data['email'],
7         password=hsh,
8         name=data['name'],
9         surname=data['surname'],
10        role=CUSTOMER
11    )
12
13 ## -- database.py -- ##
14 def add_instance(model, **kwargs):
15     try:
16         instance = model(**kwargs)
17         db.session.add(instance)
18         commit_changes()
19     except:
20         rollback_changes()
21
22 def begin_transaction():
23     db.session.begin()
24
25 def commit_changes():
26     db.session.commit()
```

```
27
28 def rollback_changes():
29     db.session.rollback()
```

Listing 22: add_instance

Tuttavia ci sono casi in cui è stato necessario fare un inserimento doppio, ad esempio quando viene fatta una prenotazione per uno slot, viene prima fatto un INSERT sulla tabella 'Reservations' e poi sulla tabella 'Weight_room_reservations' che ha una FK che punta alla tabella padre 'Reservations'. Per fare entrambi gli insert in un'unica transazione, viene fatto in ordine:

1. 'add_instance_no_commit', quindi un INSERT senza commit sulla **prima tabella** in quanto puntata tramite FK NOT NULL dalla seconda tabella.

```
1 def add_instance_no_commit(model,
2     **kwargs):
3     instance = model(**kwargs)
4     db.session.add(instance)
```

Listing 23: add instance without commit

2. Viene fatto un 'flush()', altrimenti si rischierebbe che la seconda tabella non riesca ancora a vedere l'insert appena fatto in quanto quest'ultimo sarebbe ancora nel buffer in attesa di un commit.

```
1 def flush():
2     db.session.flush()
```

Listing 24: flush() method

3. 'add_instance_no_commit', sulla **seconda tabella** che quindi non ha problemi di foreign key non esistenti in quanto abbiamo fatto il flush()

4. Infine viene fatto un commit().

```
1 def commit_changes():
2     db.session.commit()
```

Listing 25: example commit

In questo modo siamo sicuri che se il primo o il secondo insert vanno in errore, verra fatto un rollback senza che siano committati dati sporchi a db.

```
1 reservation_id = str(uuid.uuid4())
2 database.add_instance_no_commit(
3     Reservations,
4     id=reservation_id,
5     customer=body['idUser'])
6 database.flush()
7 database.add_instance_no_commit(
8     WeightRoomReservations,
9     reservation_id=reservation_id,
10    slot=(body['idSlot']))
11 database.commit_changes()
```

Listing 26: example

Stesso procedimento viene usato per gli inserimenti delle prenotazioni delle lezioni legate ai corsi.

8.5 Viste

Per agevolare l'accesso ai dati e la loro visualizzazione, vengono generate delle viste subito dopo l'inizializzazione dei models. E' stato scelto di usare viste normali e non materealizzate in quanto abbiamo dati in costante aggiornamento e crescita.

1. La prima vista conta, per ogni slot, il suo numero di prenotazioni corrente, cio'è quante persone sono iscritte a quel slot in questo momento, facendo un "count(*)" sulla tabella delle prenotazioni riguardanti la sala pesi, in questo modo a front-end riusciamo a stampare sull'calendario tutti gli slot con il corrispettivo numero di iscritti senza problemi.

```

1 DROP VIEW IF EXISTS gym.
  slots_with_current_reservation_V
  ;
2 CREATE VIEW gym.
  slots_with_current_reservation_V
  AS
3 SELECT s.*, count(w.*) as
  current_reservations
4 FROM gym.slots s left join gym
  .weight_room_reservations w on
  s.id = w.slot
5 group by id, date, time_from,
  time_to, max_capacity;
```

Listing 27: prima vista

2. La seconda vista sfrutta la prima vista per stampare tutti gli slot che sono prenotabili, cio'è che hanno lo spazio disponibile almeno per una persona, in questo modo con una semplice query, prima di fare una prenotazione per la sala pesi, possiamo verificare se lo slot interessato è al limite della capienza o no.

```

1 DROP VIEW IF EXISTS gym.
  there_Is_Space_In_Slot_View;
2 CREATE VIEW gym.
  there_Is_Space_In_Slot_View AS
3 SELECT slotReservations.id
  FROM gym.
  slots_with_current_reservation_V
  as slotReservations
4 where max_capacity>
  current_reservations;
```

Listing 28: seconda vista

3. La terza vista è simile alla prima, viene usata per contare, per ogni lezione, il suo numero di prenotazioni corrente, cio'è

quante persone sono iscritte a quella determinata lezione in questo momento, facendo un "count(*)" sulla tabella delle prenotazioni riguardanti i corsi, in questo modo a front-end riusciamo a stampare sull'calendario tutte le lezioni con il corrispettivo numero di iscritti senza problemi.

```

1 DROP VIEW IF EXISTS gym.
  lessons_with_current_reservation_V
  ;
2 CREATE VIEW gym.
  lessons_with_current_reservation_V
  AS
3 SELECT l.id, l.date, l.time, l
  .max_participants, count(lr.*)
  as current_reservations, c.name
  as course, c.description as
  course_description
4 FROM gym.lessons l
5 left join gym.
  lesson_reservation lr on l.id =
  lr.lesson
6 inner join gym.courses c on c.
  id = l.course
7 group by l.id, date, time,
  max_participants, c.name, c.
  description;
```

Listing 29: terza vista

4. La quarta vista sfrutta la terza vista per stampare tutte le lezioni che sono prenotabili, cio'è che hanno lo spazio disponibile almeno per una persona, in questo modo con una semplice query, prima di fare una prenotazione per una lezione relativa a un corso, possiamo verificare se la lezione interessata è al limite della capienza o no.

```

1 DROP VIEW IF EXISTS gym.
  there_Is_Space_In_Lesson_View;
2 CREATE VIEW gym.
  there_Is_Space_In_Lesson_View
  AS
3 SELECT lessonReservations.id
4 FROM gym.
  lessons_with_current_reservation_V
  as lessonReservations
5 where max_participants>
  current_reservations;
```

Listing 30: quarta vista

5. La quinta vista si presta a mostrare per ogni utente tutte le sue prenotazioni, sia legate alle lezioni che alla sala pesi, viene sfruttata nell'area riservata per tenere traccia dello storico o prenotazioni pianificate.

```

1 DROP VIEW IF EXISTS gym.
  user_With_All_Reservations;
2 CREATE VIEW gym.
  user_With_All_Reservations AS
```



```

3      SELECT 'lesson' as
      reservation_type,r.id,r.date,r.
      time,r.customer,l.
      reservation_id,l.lesson as slot
4      FROM gym.reservations r
5      RIGHT JOIN gym.
      lesson_reservation l on r.id =
      l.reservation_id
6 UNION ALL
7      SELECT 'weightroom' as
      reservation_type,* FROM gym.
      reservations r
8      RIGHT JOIN gym.
      weight_room_reservations on r.
      id = weight_room_reservations.
      reservation_id;

```

Listing 31: quinta vista

8.6 Indici

Per scelte progettuali, non sono stati usati indici. I motivi principali di questa scelta sono: ciascun indice costruito su una tabella rende le operazioni di inserimento, cancellazione e aggiornamento più costose, dato che anche l'indice deve essere aggiornato, nel nostro caso non conviene in quanto le tabelle sulle quali andrebbero messi gli indici sono anche quelle sulle quali vengono fatte più operazioni di insert/update. Le tabelle inoltre sono piccole e occupano un numero ridotto di pagine in memoria.

9 Ruoli e Permessi

Oltre ai ruoli a livello di entità nella WebApp, sono stati implementati dei ruoli a livello di DBMS corrispondenti appunto ai ruoli ricoperti dagli utenti a livello applicativo. La soluzione è stata ideata nel seguente modo per rispettare il principio del minimo privilegio.

```

1 CREATE USER customer WITH password '
  customer';
2 CREATE USER trainer WITH password '
  trainer';
3 CREATE USER machine WITH password '
  machine';
4 CREATE USER manager WITH password '
  manager';

```

1. Ruolo Customer: Corrisponde a un cliente della palestra. Gli verranno assegnati i permessi per poter effettuare prenotazioni a sala pesi o corsi
2. Ruolo Trainer : Corrisponde a un dipendente della palestra, con la necessità di gestire tutti i corsi. Gli verranno quindi assegnati dei permessi per poter agire sulle lezioni con operazioni di insert update e delete. Inoltre gli verranno assegnati i permessi di un utente Customer.

3. Ruolo Manager : Corrisponde al manager della palestra. Gli verranno assegnati tutti i permessi di lettura e scrittura su tutte le tabelle del DB.
4. Ruolo admin/postgres: Viene utilizzato per operazioni di sviluppo. Possiede i privilegi totali, anche sulle modifiche dello schema.
5. Ruolo machine: Corrisponde alla macchinetta che agevola l'accesso ai customer all'entrata della palestra che quindi apre lo sportello al passaggio della tessera. Quindi ha i permessi di visualizzare la lista di utenti con il loro abbonamento, e a fare operazioni di insert sulla tabella Accessi.

```

1 GRANT USAGE ON SCHEMA "gym" TO
  customer;
2 GRANT USAGE ON SCHEMA "gym" TO trainer
  ;
3 GRANT USAGE ON SCHEMA "gym" TO machine
  ;
4 GRANT USAGE ON SCHEMA "gym" TO manager
  ;

```

Listing 32: Schema

```

1 GRANT SELECT ON TABLE "gym".courses TO
  customer;
2 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".courses TO trainer;
3 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".courses TO manager;

```

Listing 33: Courses

```

1 GRANT SELECT ON TABLE "gym".courses TO
  customer;
2 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".lessons TO trainer;
3 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".lessons TO manager;

```

Listing 34: Lessons

```

1 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".reservations TO
  customer;
2 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".reservations TO
  trainer;
3 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".reservations TO
  manager;

```

Listing 35: Reservations

```

1 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".lesson_reservation
  TO customer;
2 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".lesson_reservation
  TO trainer;
3 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".lesson_reservation
  TO manager;

```

Listing 36: Lesson Reservations

```

1 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".
    weight_room_reservations TO
    customer;
2 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".
    weight_room_reservations TO
    trainer;
3 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".
    weight_room_reservations TO
    manager;

```

Listing 37: WeightRoom Reservations

```

1 GRANT SELECT ON TABLE "gym".slots TO
  customer;
2 GRANT SELECT ON TABLE "gym".slots TO
  trainer;
3 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".slots TO manager;

```

Listing 38: Slots

```

1 GRANT SELECT ON TABLE "gym".accesses
  TO customer;
2 GRANT SELECT ON TABLE "gym".accesses
  TO trainer;
3 GRANT INSERT, UPDATE ON TABLE "gym".
  accesses TO machine;
4 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".accesses TO manager
  ;

```

Listing 39: Accesses

```

1 GRANT SELECT ON TABLE "gym".policies
  TO customer;
2 GRANT SELECT ON TABLE "gym".policies
  TO trainer;
3 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".policies TO manager
  ;

```

Listing 40: Policies

```

1 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".users TO customer;
2 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".users TO trainer;
3 GRANT SELECT ON TABLE "gym".users TO
  machine;
4 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".users TO manager;

```

Listing 41: Users

```

1 GRANT UPDATE, SELECT ON TABLE "gym".
  subscriptions TO customer;
2 GRANT UPDATE, SELECT ON TABLE "gym".
  subscriptions TO trainer;
3 GRANT SELECT ON TABLE "gym".
  subscriptions TO machine;
4 GRANT DELETE, INSERT, UPDATE, SELECT
  ON TABLE "gym".subscriptions TO
  manager;

```

Listing 42: Subscriptions

10 SQL Injections

La maggior parte delle query sono state costruite con i metodi appartenenti alla libreria SQLAlchemy, che sanitizza in automatico i parametri che gli passiamo!

```

1 data = model.query.filter_by(id=id).
  first()
2 {...}
3 data = model.query.filter_by(role=role
  ).all()
4 {...}
5 data = model.query.filter_by(email=
  email).first()

```

Listing 43: ORM sanitization example

Tuttavia, per poter interrogare le viste create in precedenza, è stato necessario usare in modo grezzo e testuale le query sql, eseguendole tramite il comando `execute`. Per evitare sql injection in questo caso, è stato usato il parameter binding offerto da SQLAlchemy:

```

1 sql_query = sqlalchemy.text("select *
  from gym.
    there_Is_Space_In_Lesson_View
  WHERE id=:lessonId")
2 sql_query = sql_query.bindparams(
  bindparam('lessonId', value=
    lessonId, type_=String))
3 result = perform_query_txt(sql_query)
4 {...}
5 sql_query = sqlalchemy.text("select *
  from gym.
    user_With_All_Reservations where
    customer=:userId")
6 sql_query = sql_query.bindparams(
  bindparam('userId', value=userId,
    type_=String))
7 result = perform_query_txt(sql_query)

```

Listing 44: Parameter Binding

11 Fun code blocks

In questa sezione viene ricoperta un'interessante parte della codebase di cui vale la pena andare a parlare.

11.1 Register

Nell'istante in cui un utente si registra al sito e inserisce email e password, quest'ultima non viene salvata in chiaro sulla base di dati, ma viene hashata con una funzione di libreria che genera l'hash della password con l'aggiunta di un salt. In questo modo nessuno potrà mai riconoscere le effettive password degli utenti in quanto non si vedranno mai in chiaro in alcuna parte dell'architettura.

```

1 hash = gen_pwd(data['pwd'], 'sha256')

```

Listing 45: hash password

11.2 Login

Al momento dell'autenticazione, il controllo tra la password inserita e la password salvata nel DB viene fatto tramite la funzione

```
1 if check_pwd_hash(user.pwd, your_pwd):
2     #<pwd is correct>
```

Listing 46: check password

E' stata sviluppata una comodissima funzione che permette velocemente di controllare se l'utente che sta attualmente chiamando una rotta HTTP si e' precedentemente loggato correttamente. (quindi ricevuto un token JWT dal server in seguito ad una call/login)

```
1 def ifLogged(f):
2     user = get_current_user(request)
3     return f(user) if user is not None
4     else USER_NOT_LOGGED
```

Listing 47: ifLogged

La funzione "ifLogged" prende come parametro una funzione f, e la esegue passandole come argomento un oggetto user, propriamente recuperato dal database, se e solo se get_current_user restituisce un utente, altrimenti restituisce un msg d'errore comune che indica che il chiamante non e' autenticato.

```
1 def get_current_user(request):
2     try:
3         head = request.headers
4         token = get_token(head)
5         data = jwt.decode(token,s)
6         id = data['id']
7         return get_by_id(Users,id)
8     except:
9         return None
```

Listing 48: current user

La funzione "get_current_user" si occupa quindi di prendere il token dai headers della chiamata, decodificarlo usando i secrets, recupera l'id dal jwt decodificato e lo usa per interrogare il db e ricevere gli altri campi dell'utente.

```
1 def get_token(headers):
2     return headers['x-access-token']
```

Listing 49: get token

Il jwt token viene quindi passato tramite l'header 'x-access-token'.

11.2.1 Role Check

Abbiamo inventato un modo per poter riutilizzare la stessa funzione 'ifLogged' per controllare anche se l'utente attualmente loggato ha il ruolo richiesto, ad esempio :

```
1 def ifHasRole(f, role):
2     return ifLogged(lambda user: f(
3         user))
```

```
3     if has_role(user, role)
4     else USER_NOT_AUTHORIZED)
```

Listing 50: ifHasRole

Puo' essere quindi usato per controllare se l'utente e' un admin, manager o trainer :

```
1 def ifAdmin(f):
2     return ifHasRole(f, ADMIN)
3
4 def ifManager(f):
5     return ifHasRole(f, MANAGER)
6
7 def ifTrainer(f):
8     return ifHasRole(f, TRAINER)
```

Listing 51: ifHasRole

Come verranno usate quindi queste funzioni?

```
1 def get(self):
2     return ifLogged(lambda user:
3         sendResponse(parse_me(user),""))
```

Listing 52: ifLogged

Quindi, ifLogged e passiamo una funzione lambda come paramatro che richiede un user come argomento (l'utente autenticato). In quel caso, verra' restituito sendResponse(...).

Medesima cosa con le funzioni di controllo ruolo :

```
1 def post(self):
2     return ifAdmin(lambda user:
3         sendResponse({}, "ok"))
4
```

Listing 53: ifAdmin

11.3 Response

Qualsiasi risposta inviata (tranne una sulle immagini profilo che vedremo piu' avanti) restituiscono la risposta wrappata in un oggetto appaositamente creato chiamato "Response".

```
1 def sendResponse(payload,msg,status):
2     r = Response()
3     r.data = payload
4     r.message = msg
5     r.status = status
6     return r
```

Listing 54: sendResponse

In questo modo, lato front-end possiamo sempre dare per scontato che il back-end ci restituira' un oggetto di tipo Response, indipendentemente dal dato di ritorno.

```
1 export interface Response<T> {
2     data: T;
3     message: string;
4     status: number;
5 }
```

Listing 55: Response interface

In questo modo lato front-end, nell'apiService qualsiasi get/post necessaria si potra' sempre fare usando Response come wrapper :

```

1  get<T>(url: string) {
2      return http.get<Response<T>>(
3          SERVER + url);
4  }
5
6  post<T>(url: string, body: any) {
7      return http.post<Response<T>>(
8          SERVER + url, body);
9  }

```

Listing 56: usage of Response on front-end

Grazie a questa scelta implementativa siamo riusciti a ridurre il numero di funzioni "chiamanti" da N (dove N e' il numero di rotte totali) a 2 : una per le get, e una per le post.

Piu' nel dettaglio lato angular abbiamo :

```

1  //single return
2  getAndMap<T extends Returnable<T>>(
3      url: string) {
4      return get<T>(url).pipe(map(
5          response => response.data
6      ));
7  }
8  //multiple return
9  getMAndMap<T extends Returnable<T>>(
10     url: string) {
11     return get<T[]>(url).pipe(map(
12         response => response.data
13     ));
14 }
15 // Returnable interface defined as
16 export interface Returnable<T>{
17     parse(r:Response):T;
18 }

```

Listing 57: usage of Response on front-end

Un'esempio di utilizzo della funzione getMAndMap vista prima puo' essere :

```

1  // apiservice.ts
2  public getAllUsers() {
3      updateToken();
4      return getMAndMap<User>(
5          '/users/all',
6          httpOptions
7      );
8  }
9  // user.ts
10 export class User implements
11     Returnable<User> {
12     id: string = "";
13     name: string = "";
14     birth_date!: Date;
15     ...
16 }

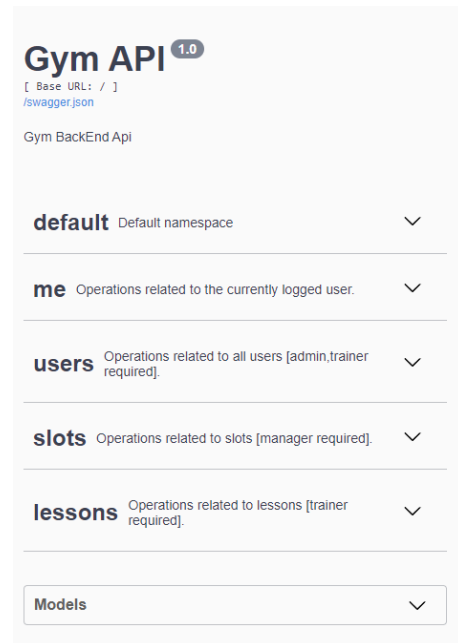
```

Listing 58: usage of getMAndMap

Affascinante.

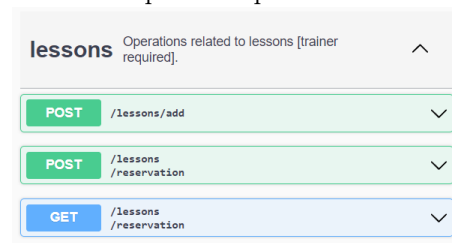
12 Swagger e Documentazione

Swagger e' un tool che e' stato utilizzato assieme a flask_restx per mappare di fianco al codice backend la documentazione sulle api sviluppate. Il risultato finale si presenta come pagina html nella quale si puo' comodamente vedere in tempo reale le api offerte dal servizio (up and running) e una piccola console dove si possono provare.

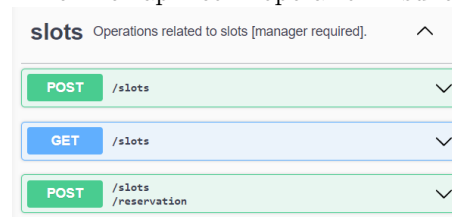


Le diverse rotte sono state raggruppate in 5 namespaces. Ogni namespace condivide un baseurl comune, alla quale viene concatenato un url_path :

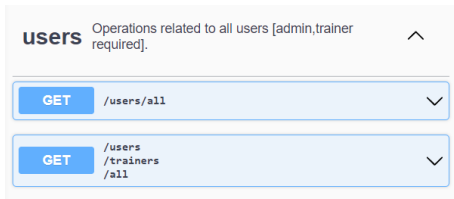
Per le api con operazioni sulle Lessons:



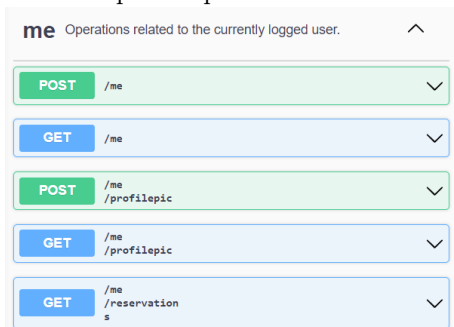
Per le api con operazioni sulle Slots:



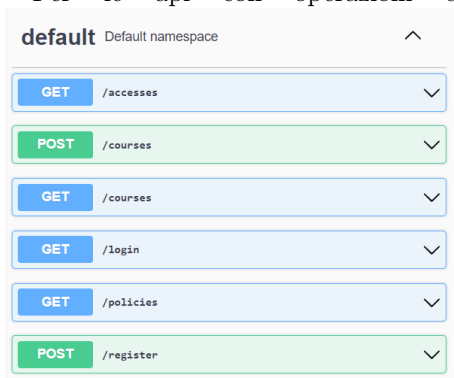
Per le api con operazioni sui Users:



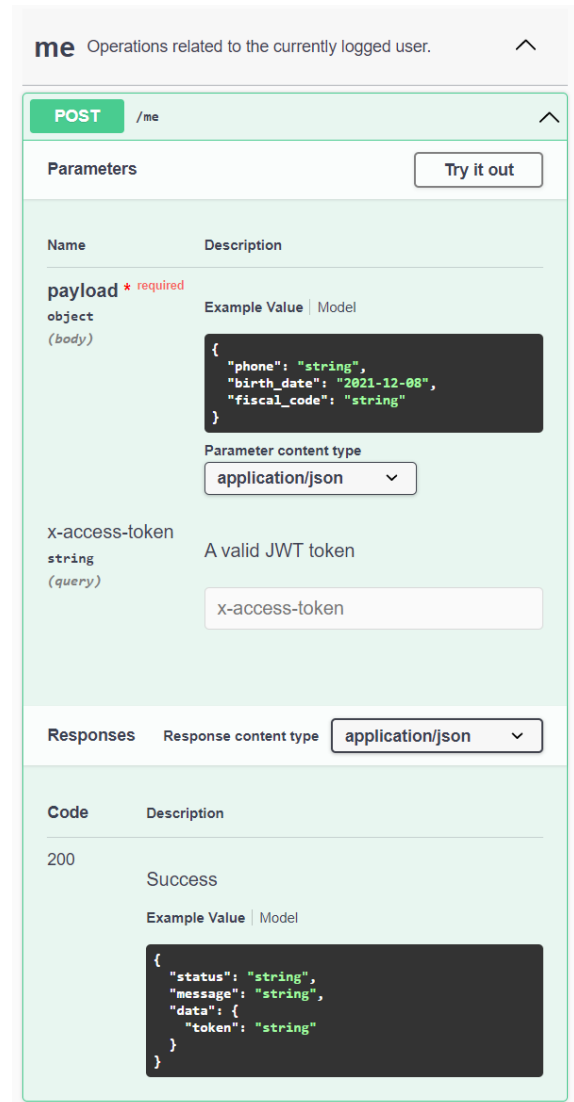
Per le api con operazioni sull'utente loggato:



Per le api con operazioni default:



Per ognuno di questi blocchi si può facilmente vedere il payload format expected e il payload format di ritorno.



Si può inoltre sempre tramite la stessa interfaccia provare l'api con il tasto Try-ItOut>Execute.

Curl

```
curl -X 'GET' \
'http://vps-487579d2.vps.ovh.net:5000/slots' \
-H 'accept: application/json'
```

Request URL

```
http://vps-487579d2.vps.ovh.net:5000/slots
```

Server response

| Code | Details | | | | |
|------|---|------|-------------|-----|---------|
| 200 | <p>Response body</p> <pre>{ "data": [{ "current_reservations": 0, "date": "2021/12/21", "description": "Descrizione slot", "id": "74708543-76c8-41c8-90ba-eb09dc9ee362", "max_capacity": 20, "time_from": "12:00:00", "time_to": "16:00:00", "title": "Titolo Slot" }, { "current_reservations": 0, "date": "2021/12/21", "description": "Descrizione slot", "id": "825a4516-0827-4030-8809-1e30cfd0f8bf", "max_capacity": 30, "time_from": "16:00:00", "time_to": "20:00:00", "title": "Titolo Slot" }, { "current_reservations": 0, "date": "2021/12/10", "description": "Descrizione slot", "id": "825a4516-0827-4030-8809-1e30cfd0f8bf", "max_capacity": 30, "time_from": "16:00:00", "time_to": "20:00:00", "title": "Titolo Slot" }] }</pre> <p>Response headers</p> <pre>access-control-allow-headers: Authorization,Content-Type access-control-allow-methods: POST access-control-allow-origin: * content-length: 2109 content-type: application/json,application/json date: Wed,08 Dec 2021 22:39:14 GMT server: Werkzeug/2.0.2 Python/3.8.12</pre> <p>Responses</p> <table border="1"> <thead> <tr> <th>Code</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>200</td> <td>Success</td> </tr> </tbody> </table> | Code | Description | 200 | Success |
| Code | Description | | | | |
| 200 | Success | | | | |

Questo ci ha permesso di avere sempre una versione aggiornata della documentazione delle api mano a mano che lo sviluppo proseguiva, inoltre offre un'alternativa a PostMan.

Ma come funziona lato flask?

Vediamo ad esempio le rotte relative all'utente loggato :

```
1 @me_ns.route('/')
2 @me_ns.param('x-access-token', 'A
  valid JWT token')
3 class Me(Resource):
4
5 @me_ns.marshall_with(mess_of(user_api))
6 def get(self):
7     return ifLogged(lambda user:
8         sendResponse(parse_me(user), ""))
9
10 @me_ns.expect(update_user_api)
11 @me_ns.marshall_with(mess_of_string())
12 def post(self):
13     return ifLogged(lambda user:
14         doFinallyCatch(lambda:
15             update_me(user, request),
```

```
sendResponse({}, "Updated"),
sendResponse({}, "Error")
))
)
```

Listing 59: usage of swagger into flask

Ogni gruppo di api quindi viene mappato come classe python, e tramite le annotation `routeparammarshal_withexpect` andiamo a definire sia l'api che il comportamento generale del nostro applicativo.

Particolare attenzione va prestata all'oggetto base `@ms_ns` che indica il namespace creatosi per questo set di rotte :

```
1 me_ns = Namespace('me',
2 description='Operations...')
3
4 api.add_namespace(me_ns)
```

Listing 60: usage of swagger into flask

Oltre ad `ms_ns` sono stati creati in modo simile anche tutti gli altri namespace.

Interessante notare come viene impostato il tipo di ritorno del metodo HTTP GET `/me` ->

```
1 # ... code before ...
2
3 @me_ns.marshall_with(mess_of(user_api))
4 def get(self):
5     return ifLogged(lambda user:
6         # ... code after ...
```

Listing 61: usage of swagger into flask

Data la scelta implementativa di wrappare tutti gli oggetti di ritorno all'interno di un oggetto `Response`, abbiamo ora la possibilita' di creare una funzione `mess_of` che prende come argomento il modello dell'oggetto che intendiamo rispondere, per creare un oggetto wrapper di ritorno :

```
1 def mess_of(model):
2     return api.model('Response', {
3         'status': fields.String(),
4         'message': fields.String(),
5         'data': fields.Nested(model)
6     })
```

Listing 62: usage of swagger into flask

Cosa medesima per le api che restituiscono una lista di risultati :


```
1 def mess_ofs(model):
2     return api.model('Response', {
3         'status': fields.String(),
4         'message': fields.String(),
5         'data': fields.List(fields.
6             Nested(model))
7     })
```

Listing 63: usage of swagger into flask

13 Image Upload & Download

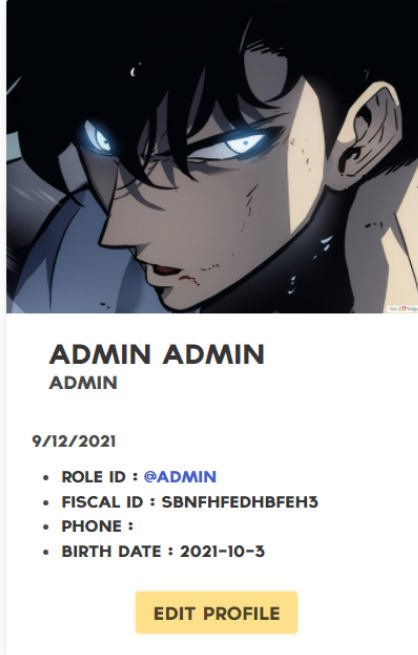
L'upload & download delle immagini e' una divertente feature che e' stata inserita per poter cambiare immagine profilo di un'utente loggato.

Essenzialmente lato front-end nell'area privata di un'utente loggato, e' possibile aggiornare alcuni campi. Tra questi vi e' anche la possibilita' di cambiare l'immagine profilo :



The screenshot shows a user profile for 'ADMIN ADMIN'. The profile picture is a muscular man with sunglasses. Below the name, the role is 'ADMIN' and the birth date is '9/12/2021'. A list of user details includes: ROLE ID: @ADMIN, FISCAL ID: SBNFBDWUWNJ3ID, PHONE: 298346348234, and BIRTH DATE: 2021-II-2. Below this, there are input fields for 'DATE OF BIRTH' (12/07/2021), 'FISCAL ID' (SBNFBDWUWNJ3ID), and 'PHONE NUMBER' (298346348234). A 'Choose File' button is next to the phone number field, and a 'SUBMIT EDIT' button is at the bottom.

Si puo' quindi cambiare immagine profilo seguendo gli step: 'Area Personale' > 'Edit Profile' > 'Choose File' > 'Submit Edit'.



The screenshot shows a user profile for 'ADMIN ADMIN'. The profile picture is a character with blue eyes. Below the name, the role is 'ADMIN' and the birth date is '9/12/2021'. A list of user details includes: ROLE ID: @ADMIN, FISCAL ID: SBNFHFDHBFH3, PHONE: , and BIRTH DATE: 2021-10-3. Below this, there is a yellow 'EDIT PROFILE' button.

Per poter gestire le immagini in questo modo e' stato necessario lo sviluppo di un'api in grado di gestire una chiamata http multipart.

Ma il nucleo della logica e' riassumibile con questi 3 metodi :

```
1 def allowed_file(filename):
2     return '.' in filename and \
3         filename.rsplit('.', 1)[1].
4         lower() in ALLOWED_EXTENSIONS
5
6 def upload_file(user_id):
7     if 'file' not in request.files:
8         return '1'
9     file = request.files['file']
10    if file.filename == '':
11        return '2'
12    if file and allowed_file(file.
13        filename):
14        file_upd = user_id
15        filename = secure_filename(
16            file_upd) + "." + file.filename.
17            rsplit('.', 1)[1].lower()
18        file.save(os.path.join(app.
19            config['UPLOAD_FOLDER'], filename)
20        )
21        return 'True'
22
23 def download_profilepic(id):
24     for ext in ALLOWED_EXTENSIONS:
25         if os.path.isfile(app.config["
26             UPLOAD_FOLDER"] + id + "." + ext):
27             return send_from_directory
28             (app.config["UPLOAD_FOLDER"], id +
29             "." + ext)
30     return None
```

Listing 64: file download upload

14 CI/CD

Un scelta focale che ha permesso una riduzione massiva dello spreco di tempo in fase di build/rilascio e' stata quella di integrare una blue/red pipeline per l'integrazione e il rilascio continuo dell'applicativo back-end. Abbiamo deciso di creare una pipeline di CI/CD solo per l'applicativo rest e non anche per il front-end in quanto quest'ultimo non e' strettamente legato ai dati, mentre il database e il back-end si, quindi e' nata la necessita di avere una versione aggiornata e persistente centralizzata a tutti i contributors del dev-team. In questo documento vedremo brevemente come e' stata sviluppata una procedura del genere e quanto tempo ha risparmiato al team di sviluppo.

14.1 GitHub Actions

L'idea e' quella di rilasciare il codice compilato ed eseguirlo lato server (in una qualsiasi macchina up-and-running nel globo) ad ogni push del codice nel branch main. Quindi come "trigger" non abbiamo un cron che ogni tot minuti fa partire una build, ma un listener che rimane in ascolto a tutti i push che vengono fatti sul repo del back-end, in particolare sul branch primario main.

Per fare cio' ci sono in rete diversi tool, noi abbiamo deciso di usare le GitHub Actions. In seguito vediamo in piu' parti la definizione della [blue-pipeline](#)

```
1 name: CI
2 on:
3   push:
4     branches: [ main ]
5   pull_request:
6     branches: [ main ]
```

Listing 65: In questa prima parte andiamo a definire quando la nostra pipeline deve avviarsi.

Nel corpo della pipeline andiamo a definire 3 step, uno si occupa al login verso il docker registry, il secondo si presta per compilare l'immagine, e l'ultimo step si prende carico di pubblicarla verso il registry cloud.

```
1 jobs:
2   build:
3     runs-on: ubuntu-latest
4     env:
5       REPO: ${ secrets.DOCKER_REPO }
6     steps:
7       - uses: actions/checkout@v2
8       - name: Login
9         run: docker login
10          -u ${ secrets.
11            DOCKER_USERNAME }}
12          -p ${ secrets.DOCKER_PWD }}
```

```
12
13   - name: Build
14     run: |
15       docker build -t $REPO:latest
16       -t
17         $REPO:${GITHUB_SHA::8} .
18   - name: Publish
19     run: docker push $REPO
```

Listing 66: body blue pipeline

L'ultimo step puo' essere avviato solo in seguito a quello precedentemente visto, di default le GitHub Actions si avviano tutte in parallelo. L'ultimo step si occupa di inviare un 'notify' al server host indicandogli che c'e' una nuova versione del progetto compilata nel Docker Hub Registry, per fare cio' utilizza un specifico webhook (che andremo a vedere piu' avanti).

```
1 redeploy:
2   name: Redeploy
3   runs-on: ubuntu-latest
4   needs: [build]
5   steps:
6     - name: Deploy
7       uses: joelwmaale/webhook-
8         action@master
9       env:
10        WEBHOOK_URL:
11          ${ secrets.
12            WEBHOOK_REDEPLOY }}
```

Listing 67: redeploy trigger

14.2 GitHub Secrets

Per comprendere il codice della [blue-pipeline](#) e' necessario avere chiaro il concetto di GitHub Secrets e capire che valore attribuire ad ognuno di essi.

In particolare i secrets possono essere viste come variabili globali relative al repository git, accessibili quindi anche dalle Actions definite tramite la sintassi

```
1 ${ secrets.nome-secret }}
```

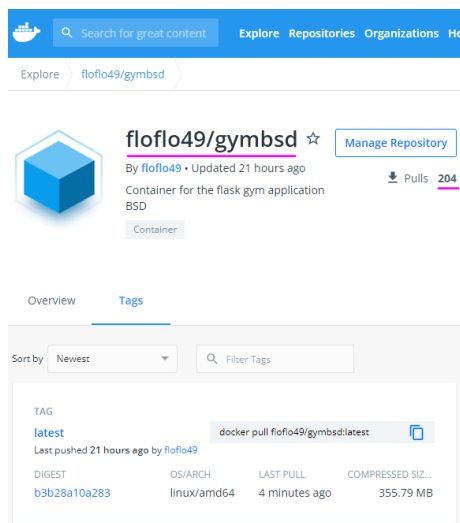
Listing 68: get-secret-git

```
1 DOCKER_REPO = <url-docker-hub-repo>
2 DOCKER_USERNAME = <user-docker-hub>
3 DOCKER_PWD = <pwd-docker-hub>
4 WEBHOOK_REDEPLOY = <http-webhook-url>
```

Listing 69: my secret env

Attenzione a non avere meta-caratteri nella pwd. (it took me 1h to figure it out)

14.3 DockerHub Registry



Come possiamo vedere dallo screenshot sopra, il numero di pull fatte a quest'immagine in totale e' di 204 ad oggi. Significa che questo sistema ci ha evitato 204 deploy manuali. Considerando che ogni deploy consiste in :

1. Build locale dell'immagine
2. Conversione dell'immagine in tar
3. Deploy del tar in ftp sul server
4. Dezip del tar
5. Load dell'immagine nel registry del vps
6. Run dell'immagine nel server in ssh

quindi in genere richiede dai 3 ai 5 minuti se si dispone di una buona connessione ad internet, significa che questa scelta ha risparmiato solo di deploy ($204 * 4$) minuti in media.

13 ore di lavoro circa sono state risparmiate, senza considerare tutti gli errori umani che si possono commettere nella fase di deploy, e altri fattori di cui non parleremo in questo documento.

14.4 WebHook Server

Un'altra parte interessante e' quella relativa alla CD, ossia la cosiddetta red-pipeline. La red-pipeline non e' altro che una sequenza di istruzioni che vengono eseguite quando la blue-pipeline ha finito il publish della nuova immagine. Alla fine della red-pipeline ci aspettiamo che nel server l'ultima versione del codice sia up and running. Affinche la vps sappia quando la blue-pipeline ha finito il suo job, mette a disposizione un rest webhook, che nel caso venga chiamato triggera un file shell interno che si occuperà del redeploy.

Per la creazione del webhook e' sufficiente il seguente comando se si lavora in ambiente linux :

```
1 webhook -hooks ./hooks.json -port 9001  
-verbose &
```

Listing 70: create-webhook.sh

Ovviamente ci sono diverse alternative, si potrebbe pensare addirittura di crearsi un proprio rest server per gestire la cosa.

Il file `hook.json` e' definito come segue :

```
1 [  
2 {  
3   "id": "reddeploy",  
4   "execute-command": "/home/ubuntu/  
5   gym/reddeploy.sh",  
6   "response-message": "Redeploying  
7   API server."  
8 }  
9 ]
```

Listing 71: create-webhook.sh

14.5 Shell Redeploy Script

Ecco quindi che viene catturata la chiamata http diretta verso il nostro server nella porta 9001 con path `/reddeploy`. E in seguito viene eseguito lo script sh : `reddeploy.sh`

```
1 #!/bin/sh  
2 docker ps -a | awk '{ print $1,$2 }' |  
3   grep <user/image> | awk '{print  
4   $1 }' | xargs -I {} docker rm -f  
5   {} && \  
6  
7 docker pull <user/image> && \  
8  
9 docker image prune -f && \  
10  
11 docker run -dit -p 5000:5000 <user/  
12 image>:latest && \  
13
```

Listing 72: create-webhook.sh

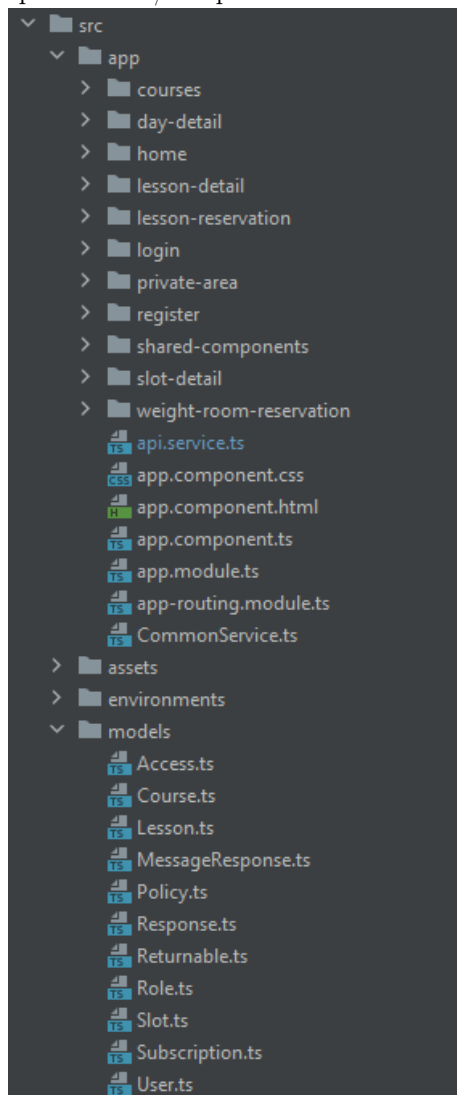
Questo e' una versione semplicissima di script di redeploy, essenzialmente rimuove il vecchio running container, scarica la nuova immagine dal repo, rimuove tutte le detach images, e infine avvia il nuovo container aggiornato.

Fondamentale la parte iniziale `#!/bin/sh`.

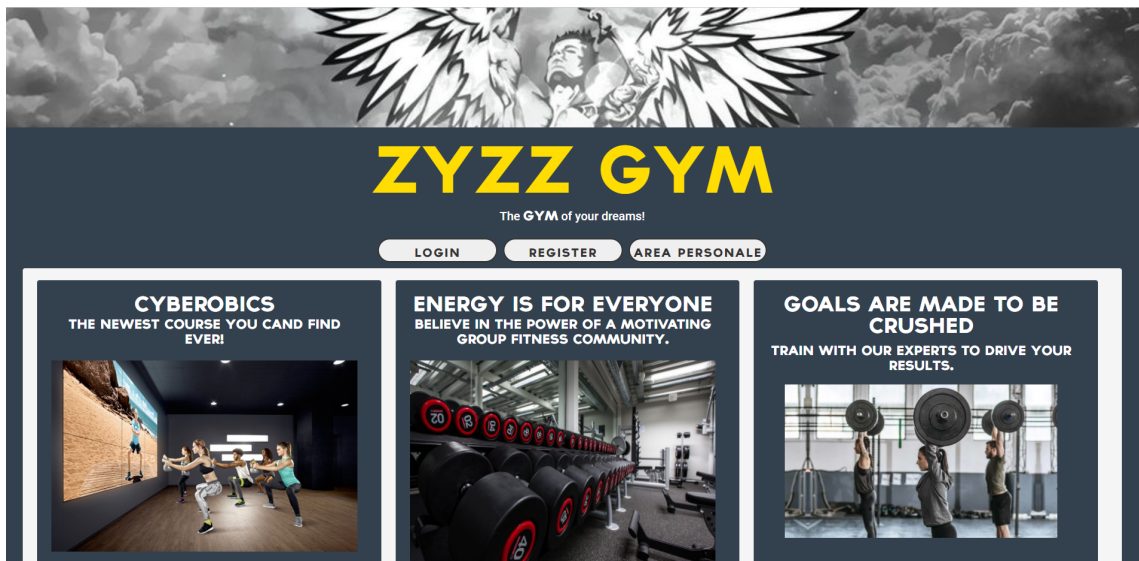
15 Componenti Angular

Per quanto riguarda il client-side del progetto abbiamo sviluppato il front-end focalizzandoci sulla creazione di una pagina web light-weight, dando la prioritá alla riduzione del codice ridondante e al riutilizzo di componenti simili. Abbiamo cercato di evitare il piu' possibile la navigazione tra piu' pagine, andando quindi sia a centralizzare la logica facendo quasi una

single-page website, sia andando a rendere l'esperienza UI/UX piu' intuitiva lato utente.



Abbiamo quindi sempre cercato di sviluppare un codice fortemente tipato usando TypeScript e Angular, definendo in modo quasi speculare i modelli lato Angular a quelli invece restituiti dall'app Flask.



OUR CURRENT COURSES
CHECK OUT OUR CURRENT AVAILABLE COURSES AND SUBSCRIBE!

ADD NEW COURSE

ADD COURSE

TITLE

DESCRIPTION

Save

CYBERROBICS
YOU ALSO HAVE TO BE LIFTING HEAVY WEIGHTS

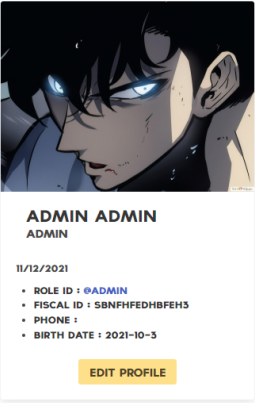
CROSSFIT
YOU ALSO HAVE TO DO CROSSFIT

☰ ☑ TODAY

SEE SLOTS SEE LESSONS

DECEMBER

| MONDAY | TUESDAY | WEDNESDAY | THURSDAY | FRIDAY | SATURDAY | SUNDAY |
|---|---|--|---|---|--|--|
| 29 NOVEMBER | 30 | 1 DECEMBER | 2 | 3 | 4 | 5 |
| 6 19:25:00 - 21:25:00 19:25:00 - 20:25:00 | 7 | 8 | 9 | 10 08:00:00 - 14:00:00 | 11 DECEMBER | 12 |
| 13 | 14 | 15 | 16 | 17 | 18 | 19 12:00:00 - CROSSFIT 13:00:00 - CROSSFIT |
| 20 | 21 12:00:00 - 14:00:00 14:00:00 - 20:00:00 08:00:00 - 12:00:00 | 22 10:00:00 - 11:00:00 10:00:00 - CROSSFIT | 23 08:00:00 - 12:00:00 12:00:00 - 14:00:00 09:00:00 - CYBEROBICS | 24 | 25 14:00:00 - 20:00:00 08:00:00 - 12:00:00 12:00:00 - 14:00:00 08:00:00 - CYBEROBICS | 26 |
| 27 11:00:00 - CROSSFIT | 28 | 29 | 30 | 31 19:25:00 - 20:25:00 20:25:00 - CYBEROBICS 11:00:00 - CROSSFIT | 1 JANUARY | 2 |



RESERVATIONS

DATE: 2021/12/04
TYPE: WEIGHTROOM
TIME: 20:00:04

DATE: 2021/12/04
TYPE: WEIGHTROOM
TIME: 20:15:45

ALL USERS :

BIRTH DATE:
EMAIL: TRAINER
NAME: XLO
ROLE: TRAINER
SURNAME: XLO

BIRTH DATE:
EMAIL: CUSTOMER
NAME: DAN
ROLE: CUSTOMER
SURNAME: IVAN

BIRTH DATE:
EMAIL: MANAGER
NAME: RLO
ROLE: MANAGER
SURNAME: RLO

BIRTH DATE: 2021/11/03
EMAIL: ADMIN
NAME: ADMIN
ROLE: ADMIN
SURNAME: ADMIN

ACCESSES

DATE: 2021/12/10
ENTRANCE: 0400:00
EXIT: 2000:00

DATE: 2021/12/11
ENTRANCE: 0400:00
EXIT: 2200:00

DATE: 2021/12/12
ENTRANCE: 0400:00
EXIT: 2200:00

POLICY

FROM: 2021/12/04
TO: 2021/12/25
DESCRIPTION: ANY
MEMBERS WHO INCURS
AN INJURY OR
BECOMES DIZZY/ILL
WHILE USING THE
GYMNESS SHOULD
IMMEDIATELY
CONTACT A FITNESS
CENTER STAFF PERSON
FOR ASSISTANCE. A
FIRST AID KIT IS KEPT
AT THE FRONT DESK
FOR MINOR INJURIES.
IN CASES REQUIRING
MORE EXTENSIVE FIRST
AID, FITNESS CENTER
STAFF WILL CONTACT
THE APPROPRIATE
PERSONS FOR
ASSISTANCE. IT IS
IMPORTANT THAT
FITNESS CENTER STAFF
BE NOTIFIED OF ANY
CASES OF INJURY OR
ILLNESS SO THAT
PROPER PROCEDURES
CAN BE INITIATED.
ATHLETIC TAPES MAY
ONLY BE USED FOR
MINOR INJURIES.

Login

Be careful

EMAIL

flo@gmail.com

PASSWORD

Login

Go Back

LOGIN

REGISTER

AREA PERSONALE

ADD LESSON

WHEN

gg/mm/aaaa

FROM

--:--

CAPACITY

0

Save

YOUR ARE LOGGED AS ROLE: ADMIN, NAME: ADMIN, EMAIL: ADMIN

ADD NEW SLOT

ADD SLOT



TITLE

 Title

DESCRIPTION

 Description

WHEN

 gg/mm/aaaa




FROM

 --:--



TO

 --:--



CAPACITY

 0

Save

16 Commits

In questa sezione del documento andremo a definire la metodologia di lavoro di gruppo che e' stata utilizzata. In particolare andremo a mostrare il numero e la frequenza di commit relative allo sviluppo del progetto front-end e di quello back-end.

| | | | |
|-------------------------------------|-----------------|----------------|---------------------|
| fixes css style | origin & master | David Ambros | Today 12:17 AM |
| fixes css style | | David Ambros | Today 12:16 AM |
| Updated Personal Area a second ti | | Ivan | Yesterday 10:36 PM |
| Changed css modal style | | David Ambros | Yesterday 10:33 PM |
| Updated Personal Area | | Ivan | Yesterday 10:19 PM |
| Added policies and accesses | | David Ambros | Yesterday 9:18 PM |
| Added policies and accesses | | David Ambros | Yesterday 9:06 PM |
| Changing api routes. | | Florian Sabani | Yesterday 4:57 PM |
| Added working Lesson Reservation | | David Ambros | Yesterday 4:26 PM |
| Removed little image from pers | | Florian Sabani | 12/7/2021 1:54 AM |
| fix user difference check on pri | | Florian Sabani | 12/7/2021 12:27 AM |
| minor fix | | David Ambros | 12/6/2021 10:29 PM |
| Added course insert + fix lesson ir | | David Ambros | 12/6/2021 10:19 PM |
| Added add Lesson but still has to l | | David Ambros | 12/5/2021 2:39 AM |
| FE : Image upload & download ac | | Florian Sabani | 12/5/2021 2:23 AM |
| Added crazy styling | | David Ambros | 12/5/2021 1:18 AM |
| FE : Start handling errors from b | | Florian Sabani | 12/5/2021 12:20 AM |
| FE : Now admin can see allUsers | | Florian Sabani | 12/4/2021 10:06 PM |
| Aggiunto qualche font + titolo + l | | David Ambros | 12/4/2021 9:36 PM |
| per ivan cosi puo vedere i ruoli | | David Ambros | 12/4/2021 8:52 PM |
| aggiunto inserimento slots | | David Ambros | 12/4/2021 8:48 PM |
| FE : Fix user update. | | Florian Sabani | 12/4/2021 8:37 PM |
| FE : Changing api routs. | | Florian Sabani | 12/4/2021 7:58 PM |
| Enhanced calendar and home but | | Ivan | 12/4/2021 6:20 PM |
| basic | | David Ambros | 12/4/2021 5:10 PM |
| FE : Finished update-user. | | Florian Sabani | 12/4/2021 1:58 PM |
| FE : minor style into edit-private | | Florian Sabani | 12/4/2021 1:03 PM |
| Aggiunto WIP di nuovo slot + vis | | David Ambros | 12/4/2021 1:10 AM |
| FE: WIP | | Florian Sabani | 12/3/2021 10:03 PM |
| FE: Fixing subscribe button probl | | Florian Sabani | 12/3/2021 9:32 PM |
| FE: Changing models fields name | | Florian Sabani | 12/3/2021 9:20 PM |
| FE: refactoring apiService and ac | | Florian Sabani | 12/3/2021 9:06 PM |
| Modified view | | David Ambros | 12/3/2021 7:40 PM |
| added first version of lessons to c | | David Ambros | 12/3/2021 7:20 PM |
| Styled home components | | Ivan | 12/3/2021 7:14 PM |
| aggiunto TODO | | David Ambros | 12/2/2021 5:46 PM |
| added things | | David Ambros | 12/2/2021 5:22 PM |
| Contenuto persolalizzato | | Florian Sabani | 12/2/2021 12:50 AM |
| Creating material dialog for day | | Florian Sabani | 12/2/2021 12:07 AM |
| Grouping slots into calendar day | | Florian Sabani | 12/2/2021 12:00 AM |
| wip | | Florian Sabani | 12/1/2021 11:37 PM |
| Area personale now makes http | | Florian Sabani | 12/1/2021 11:35 PM |
| Added Slots route | | David Ambros | 12/1/2021 2:43 AM |
| WIP Private route | | David Ambros | 11/29/2021 12:34 AM |
| Added login and private route | | David Ambros | 11/29/2021 12:10 AM |
| Created message-response-dialc | | Florian Sabani | 11/28/2021 10:42 PM |
| Completed app-register with po | | Florian Sabani | 11/28/2021 10:12 PM |
| Bind onRegister method | | Florian Sabani | 11/28/2021 9:20 PM |
| Createt input-field shared comp | | Florian Sabani | 11/28/2021 9:12 PM |
| Createt input-field shared comp | | Florian Sabani | 11/28/2021 9:10 PM |
| Added icons to register.compon | | Florian Sabani | 11/28/2021 8:55 PM |

| | | |
|------------------------|----------------|--------------------|
| B origin & main | Florian Sabani | Yesterday 9:29 PM |
| Added accesess and j | David Ambros | Yesterday 9:12 PM |
| BSD60 : Complete r | Florian Sabani | Yesterday 4:52 PM |
| FLUSHHH | David Ambros | Yesterday 4:42 PM |
| added slots insertion | David Ambros | Yesterday 4:22 PM |
| added route for less | David Ambros | Yesterday 1:46 AM |
| BSD46 : Added first | Florian Sabani | 12/7/2021 1:53 AM |
| BSD45 : Added basic | Florian Sabani | 12/7/2021 12:28 AM |
| added subrole | David Ambros | 12/6/2021 11:12 PM |
| Added course insert + | David Ambros | 12/6/2021 10:19 PM |
| Added routes for Less | David Ambros | 12/5/2021 2:40 AM |
| BSD44 : Moved uplo | Florian Sabani | 12/5/2021 2:17 AM |
| BSD44 : Add photo t | Florian Sabani | 12/5/2021 1:49 AM |
| BSD31 : Fixing requ | Florian Sabani | 12/4/2021 11:53 PM |
| BSD31 : Changing H' | Florian Sabani | 12/4/2021 11:41 PM |
| BSD31 : Fixing login | Florian Sabani | 12/4/2021 11:39 PM |
| BSD31 : Move login | Florian Sabani | 12/4/2021 11:26 PM |
| BSD31 : Refactoring | Florian Sabani | 12/4/2021 11:25 PM |
| BSD31 : Refactoring | Florian Sabani | 12/4/2021 11:05 PM |
| WIP | David Ambros | 12/4/2021 10:24 PM |
| BSD31 : Adding ifAd | Florian Sabani | 12/4/2021 10:01 PM |
| BSD31 : Introduced | Florian Sabani | 12/4/2021 9:38 PM |
| BSD30 : Refactoring | Florian Sabani | 12/4/2021 9:31 PM |
| BSD24 : Moved date | Florian Sabani | 12/4/2021 8:53 PM |
| BSD24 : Avoid passir | Florian Sabani | 12/4/2021 8:39 PM |
| BSD24 : Avoid editir | Florian Sabani | 12/4/2021 8:14 PM |
| BSD24 : Avoid editir | Florian Sabani | 12/4/2021 8:12 PM |
| BSD24 : Refactoring | Florian Sabani | 12/4/2021 7:53 PM |
| trying to fix view but | David Ambros | 12/4/2021 7:36 PM |
| added user tests | David Ambros | 12/4/2021 7:36 PM |
| trying to fix view but | David Ambros | 12/4/2021 6:24 PM |
| pushino refactoring | David Ambros | 12/4/2021 5:17 PM |
| BSD24 : Fix returnin | Florian Sabani | 12/4/2021 1:55 PM |
| BSD24 : Add more r | Florian Sabani | 12/4/2021 1:52 PM |
| BSD24 : Fix datetim | Florian Sabani | 12/4/2021 1:43 PM |
| BSD24 : Edit date in | Florian Sabani | 12/4/2021 1:39 PM |
| BSD23 : Fix /me api | Florian Sabani | 12/4/2021 1:29 PM |
| BSD23 : Changed co | Florian Sabani | 12/4/2021 1:21 PM |
| BSD23 : Fixed overk | Florian Sabani | 12/4/2021 1:18 PM |
| BSD23 : Add first UF | Florian Sabani | 12/4/2021 1:13 PM |
| BSD23 : Add also ph | Florian Sabani | 12/3/2021 9:57 PM |
| BSD22 : Changed foi | Florian Sabani | 12/3/2021 8:11 PM |
| BSD22 : Fast fix my_ | Florian Sabani | 12/3/2021 8:08 PM |
| BSD22 : I need to fb | Florian Sabani | 12/3/2021 7:57 PM |
| BSD22 : Fix query ge | Florian Sabani | 12/3/2021 7:52 PM |
| BSD22 : Fix method | Florian Sabani | 12/3/2021 7:47 PM |
| BSD22 : Add mySub: | Florian Sabani | 12/3/2021 7:39 PM |
| added fetchLessonsR | David Ambros | 12/3/2021 7:26 PM |
| added addSlotReserv | David Ambros | 12/2/2021 5:24 PM |
| BSD11 : Adding Tidu | Florian Sabani | 12/2/2021 12:38 AM |
| BSD11 : Fixing enur | Florian Sabani | 12/1/2021 11:35 PM |
| BSD11 : Add check r | Florian Sabani | 12/1/2021 11:09 PM |

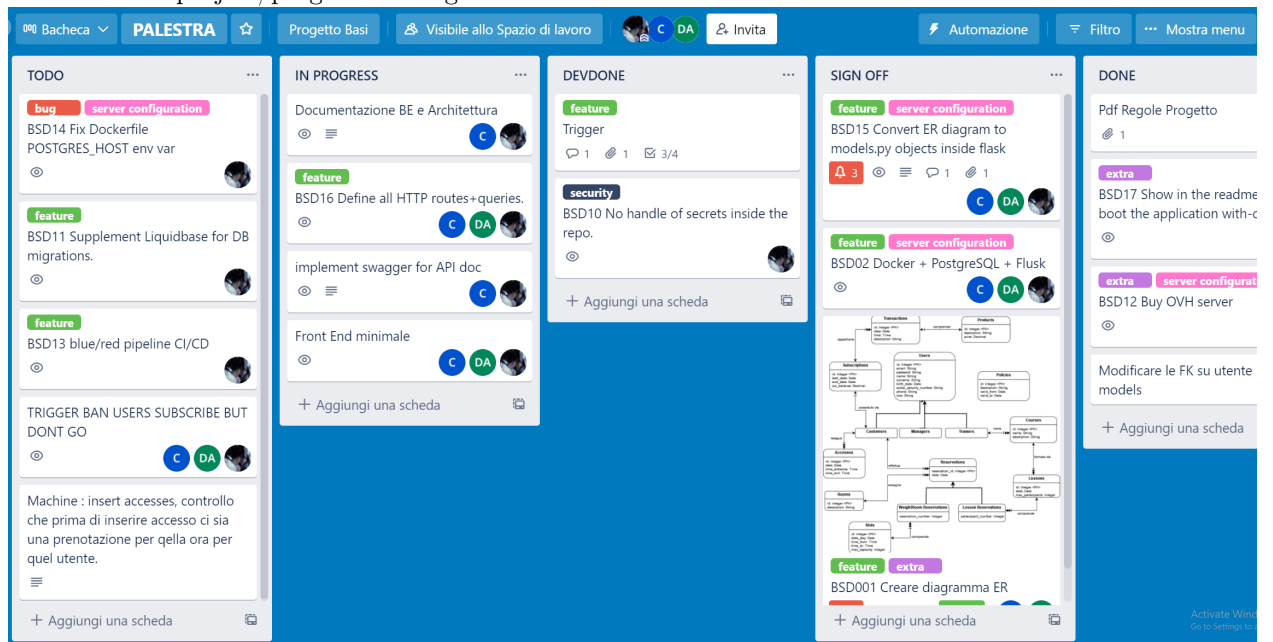
Una cosa che salta subito all'occhio e' che nessuno ha mai pushato in un branch diverso dal principale. Questo e' perche' abbiamo deciso di lavorare seguendo il principio del [trunk based development](#).

17 Trello & Agile Working

Abbiamo sempre cercato di lavorare seguendo le orme della metodologia Agile/SCRUM, cercando di separare le storie-lavoro nelle seguenti colonne :

1. TODO
2. IN PROGRESS
3. DEV-DONE
4. SIGN-OFF
5. DONE

Abbiamo usato trello come piattaforma di project/progress management.



Conclusions

TODO

Acknowledgements

TODO

References