Progetto basi di dati

Data di consegna: 19/12/2021 - sessione invernale
Nome gruppo: MCpalestreMC
Componenti: Mario Coci 876422
Scelta progetto: MCpalestreMC (scelta delle palestre)
DBMS scelto: MYSQL
Versione di Python: 3.7

Indice

1-Premessa iniziale

Per la realizzazione di questo progetto, l'applicazione è stata sviluppata con Visual Studio 2019 in Flask, utilizzando SQLAlchemy Core (Textual SQL) per interfacciarsi al DBMS sottostante (MySQL). Le pagine per la web app sono state sviluppate a parte con HTML (con l'ausilio di CSS e Javascript dove necessario) e poi renderizzate da Flask.

Il tema scelto per il progetto è: Palestra in periodo di Covid-19.

La grafica delle pagine HTML è stata realizzata attraverso il framework Bootstrap (https://getbootstrap.com/), infatti nelle pagine html ci sono dei link e riferimenti a file .css e .js di Bootstrap. Per quanto riguarda i form, ne è stato implementato l'utilizzo tramite una libreria di Python chiamata WTF(What the forms).

Prima di far avviare l'applicazione, è necessario eseguire alcune procedure:

1. Nella cartella del progetto è presente una cartella chiamata 'Database' da importare in MySQL.
2. Nella sezione 'Users and privileges' di MySQL, cliccare sull'utente 'anonimo' e su 'Administrative Role' spuntare 'MaintenanceAdmin', 'ProcessAdmin', 'UserAdmin' e 'SELECT', quindi premere 'Apply'.
3. Sempre nella sezione 'Users and Privileges', cliccare sul ruolo 'Cliente' e su 'Administrative Roles' spuntare 'UserAdmin', quindi premere 'Apply'.
4. Sempre nella sezione 'Users and Privileges', cliccare sul ruolo 'Istruttore' e su 'Administrative Roles' spuntare 'UserAdmin' e 'SecurityAdmin', quindi premere 'Apply'.
5. Sempre nella sezione 'Users and Privileges', cliccare sul ruolo 'Gestore' e su 'Administrative Roles' spuntare 'DBA', quindi premere apply.

Ora è possibile avviare l'applicazione web.

2-Esposizione di cos’è il progetto e tutte le sue funzionalità

Il progetto si basa sulla realizzazione di un’applicazione web per una compagnia di palestre. Il nome della compagnia è ‘MCpalestreMC’.

L’idea di base era quella di realizzare un sito dove si avessero alcune possibilità: prenotare una lezione, se non malato di Covid-19, registrarsi, aggiungere locali ad una palestra, consultare alcune informazioni riguardanti le palestre e i corsi…
Sono stati implementati 3 ruoli: ‘Cliente’, ‘Istruttore’ e ‘Gestore’. Ognuno di questi ruoli ha possibilità distinte di interagire con le tabelle del DBMS, secondo il principio del ‘minimo privilegio’. All’avvio del server, l’utente viene proiettato nella home-page del sito(comune a tutti i ruoli) in qualità di user anonimo. Una volta loggato, il fruitore avrà delle pagine distinte in cui potrà accedere.

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

- L’elenco delle palestre
- L’elenco dei corsi disponibili
- L’elenco degli abbonamenti

Durante tutto l’uso della web app è presente una barra in alto da cui si possono fare le seguenti operazioni:

- Accedere alla home-page
- Scoprire la storia delle palestre MCpalestreMC
- Registrarsi
- Fare login

Se loggato, il pulsante della registrazione viene disabilitato e vengono concesse, oltre alle funzioni appena citate, altre 2:

- Fare logout
- Accedere all’area riservata

La pagina di registrazione permette di inserire per la prima volta l’utente nella base di dati. Allo user sono richiesti codice fiscale, nome, cognome, email, numero, password e la palestra che intende usare. Viene inserito come tipo utente il ‘Cliente’ e ‘Negativo’ al Covid-19 settato di default, gli vengono dati i privilegi da ‘Cliente’ e poi viene spedito nell’area riservata. E’ stato scelto il codice fiscale come codice utente poiché già di per sé univoco.

La pagina di login permette l'autenticazione tramite codice fiscale e password. poi manda l'utente alla propria area riservata.

Nella pagina riservata, che è diversa per ogni utente, il cliente può:

- Modificare il numero di telefono, l'email oppure la palestra dove si allena
- Vedere i corsi a cui non è iscritto e iscriversi
- Prenotare una lezione
- Vedere le prenotazioni fatte
- Segnalare di avere il Covid-19
- Vedere e disiscriversi dai corsi che segue

L'istruttore potrà:

- Modificare il numero di telefono, l'email oppure la palestra dove insegna
- Visualizzare i corsi che tiene, il loro orario, le persone iscritte, il locale, il periodo in cui si tiene il corso, il giorno, l'orario e la descrizione; inoltre può scegliere di eliminare il corso
- Creare un nuovo corso e scegliere il locale in cui svolgerlo
- Segnalare di avere il Covid-19

All'aggiunta di un nuovo corso verranno chiesti titolo, descrizione, data di inizio corso, data di fine corso, giorno della settimana, orario e il locale in cui si svolgerà.

Il gestore, invece, potrà:

- Modificare il numero di telefono, l'email oppure la palestra
- Vedere i dettagli della sua palestra, come l'indirizzo, l'email, il numero di telefono e il numero di clienti iscritti
- Creare nuovi locali per la sua palestra
- Registrare nuovi istruttori e nuovi gestori (quindi creare nuove palestre ed i relativi locali iniziali)

Per la registrazione di istruttori e gestori, vengono richiesti gli stessi campi dei clienti, l'unica differenza è che verrà inserito un tipo utente diverso e verranno dati privilegi diversi in base al ruolo.

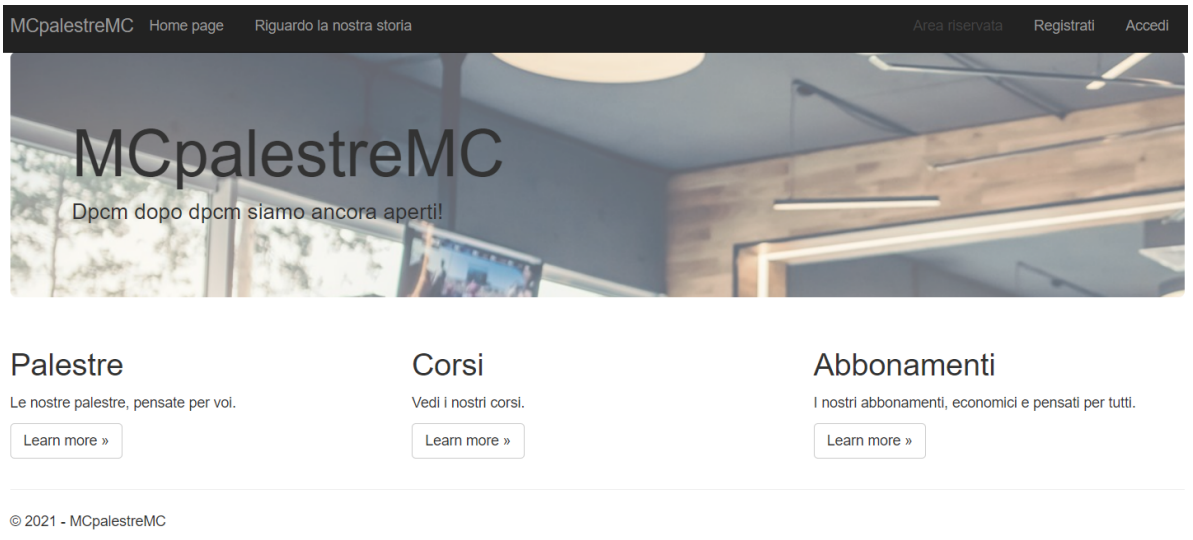
All'aggiunta di una nuova palestra verranno chiesti: nome della palestra, indirizzo della palestra, email della palestra, numero di telefono della palestra e un numero variabile di locali (a discrezione del gestore e che in futuro possono essere aumentati) per ognuno dei quali viene chiesto di indicare la metratura e la capienza di persone massima.

All'aggiunta di un nuovo locale verranno chieste metratura e capienza di persone massima.

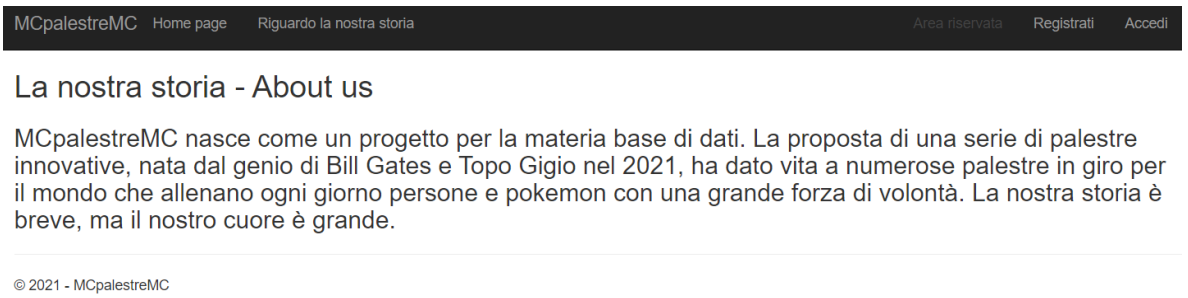
3-Schermate presenti nella web app

BARRA DI NAVIGAZIONE (login e logout)

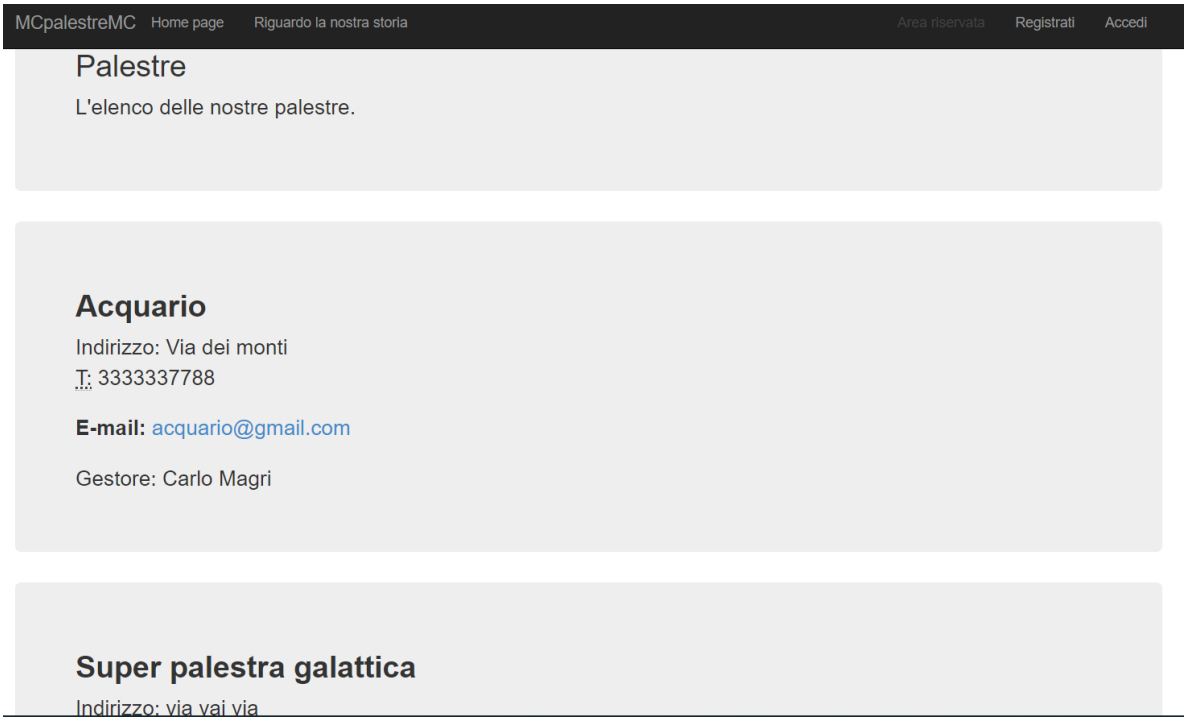
HOME PAGE



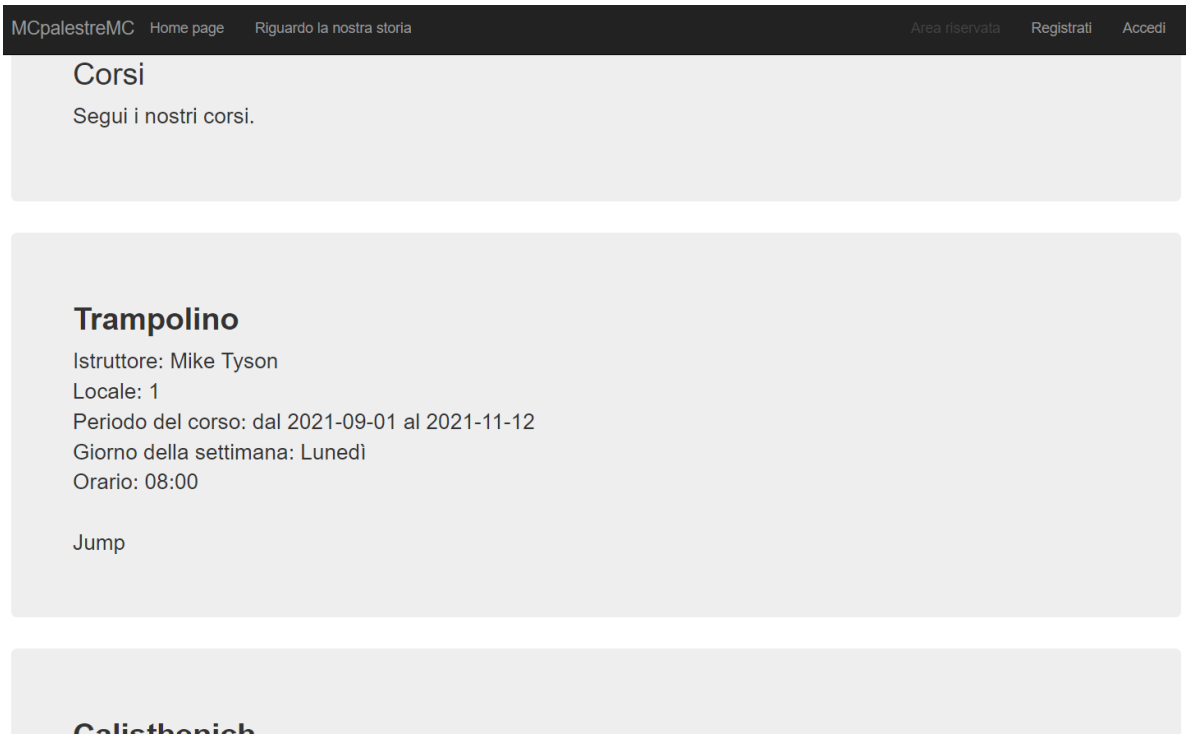
ABOUT US



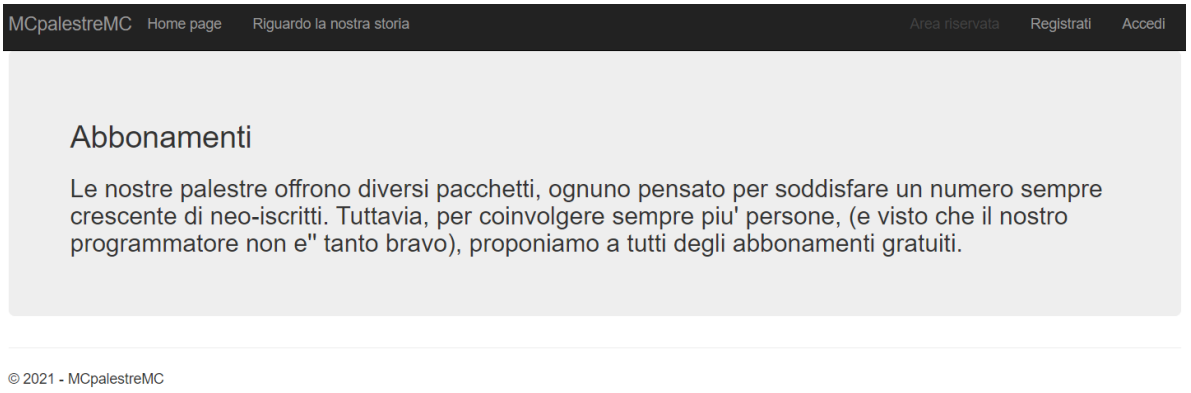
PALESTRE



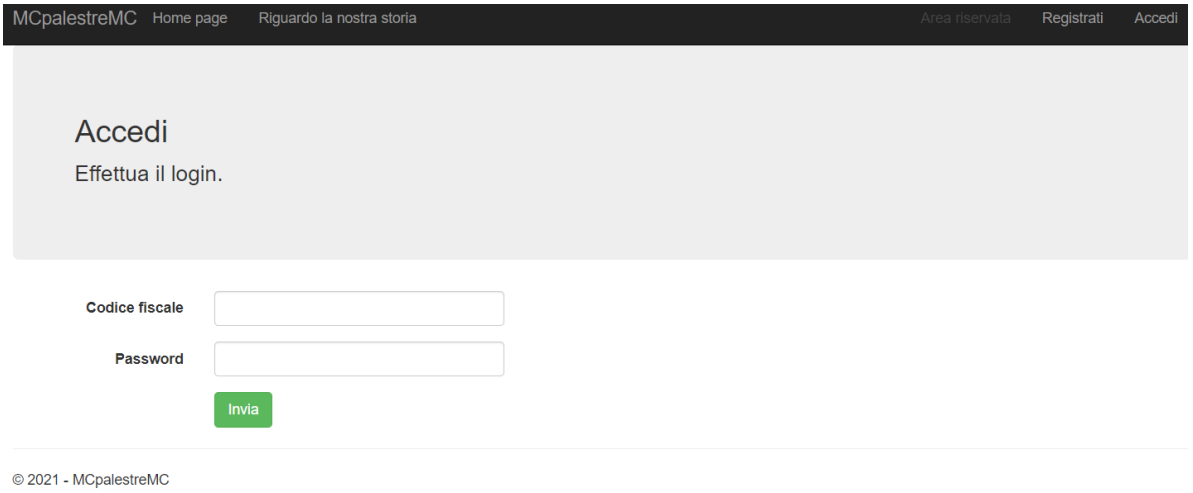
CORSI



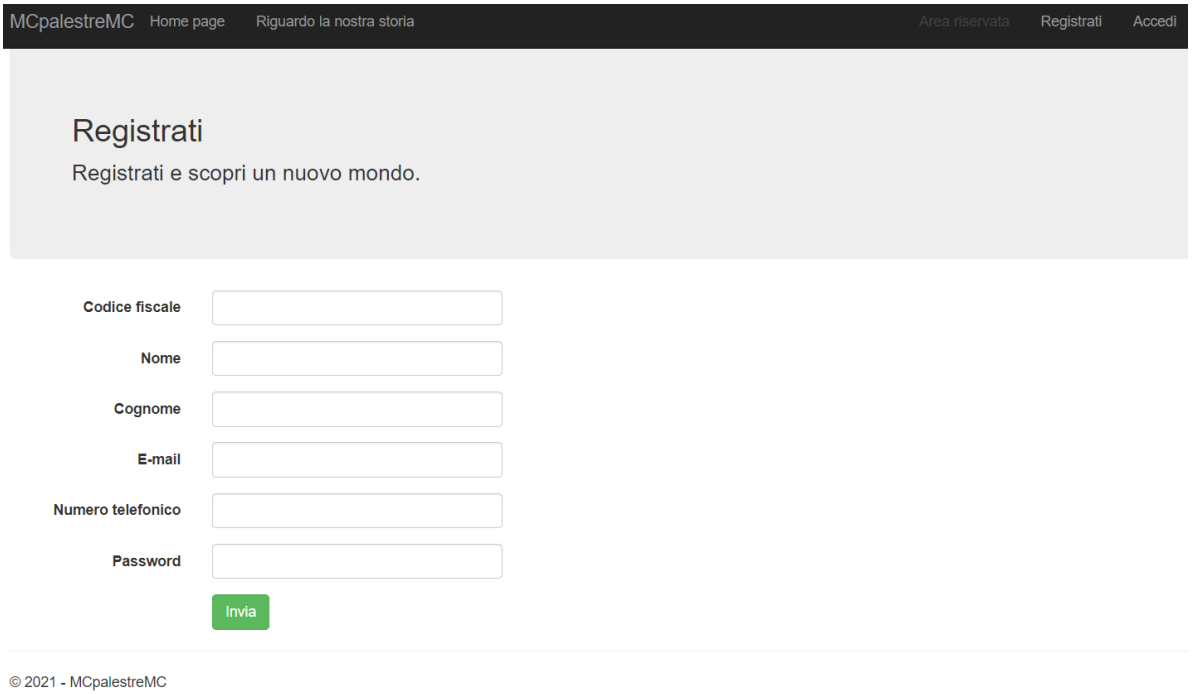
ABBONAMENTI



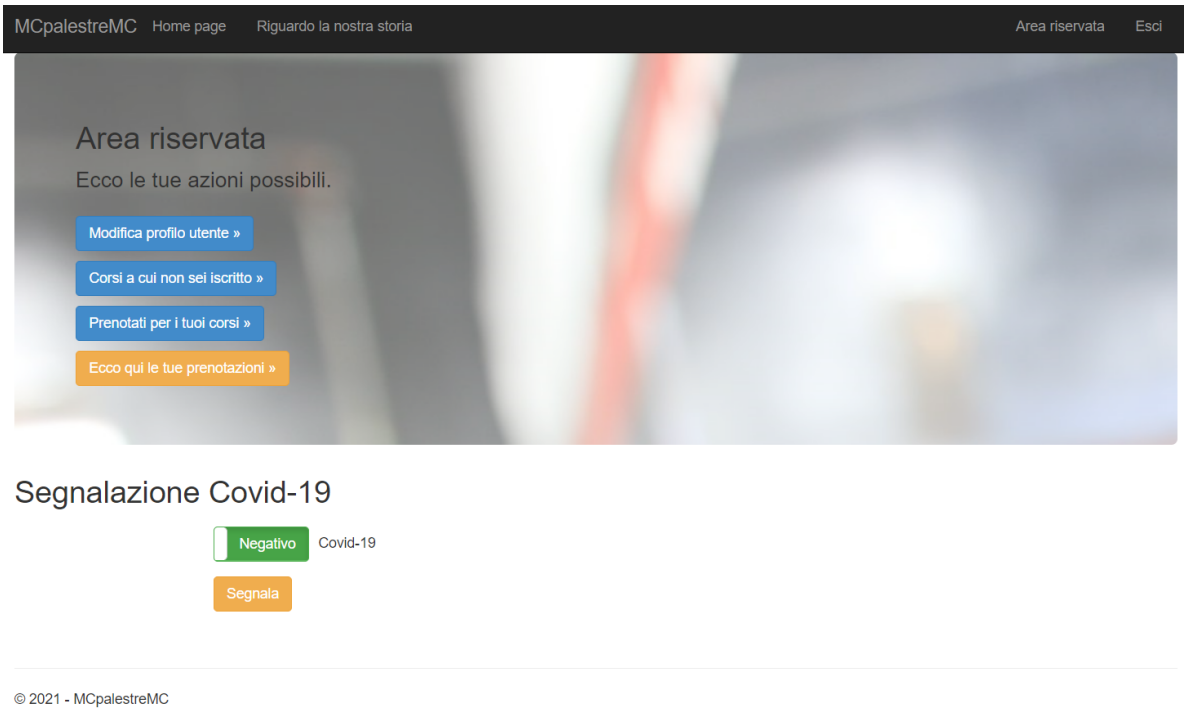
LOGIN



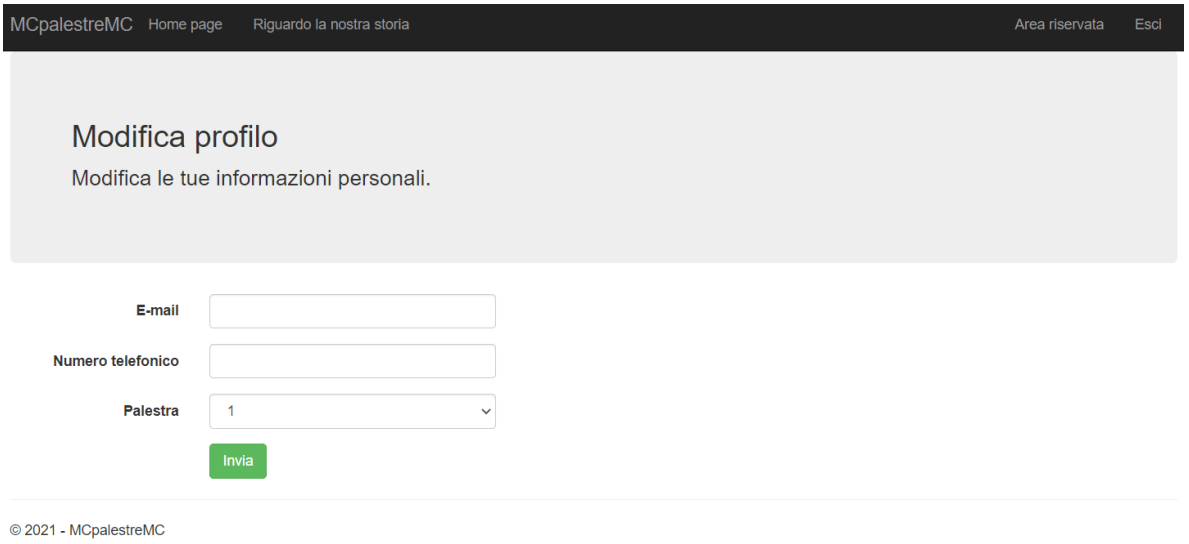
REGISTRAZIONE CLIENTE



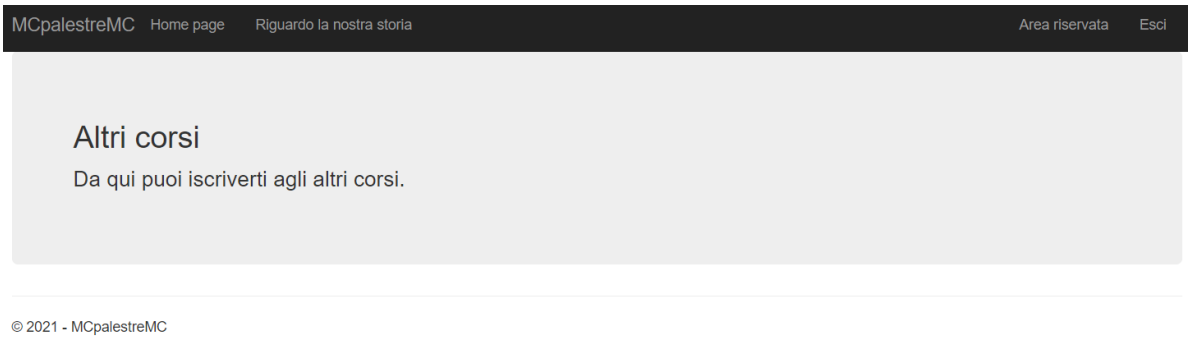
AREA RISERVATA CLIENTE - pagina principale



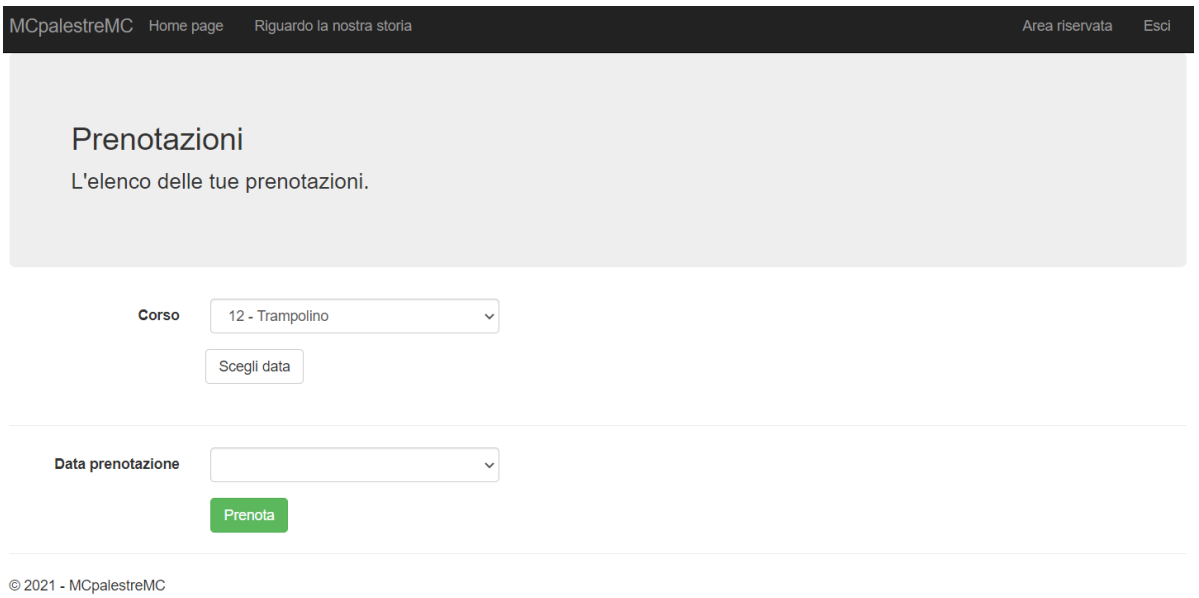
AREA RISERVATA - modifica profilo



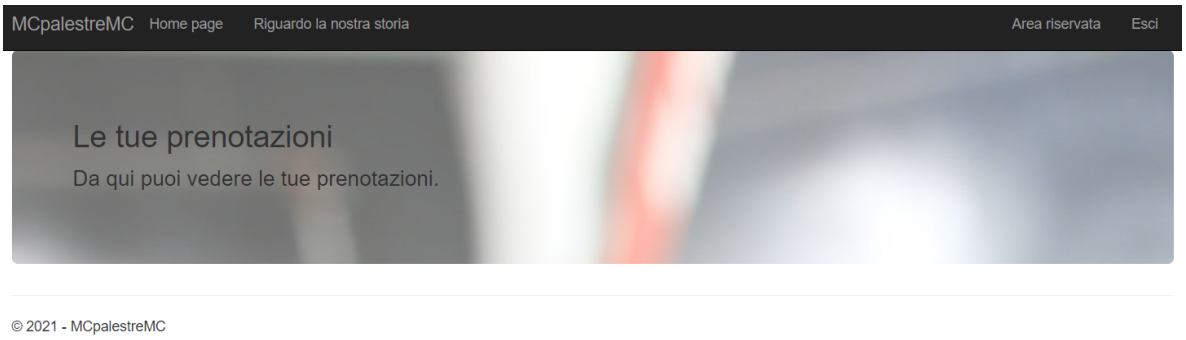
AREA RISERVATA CLIENTE - altri corsi



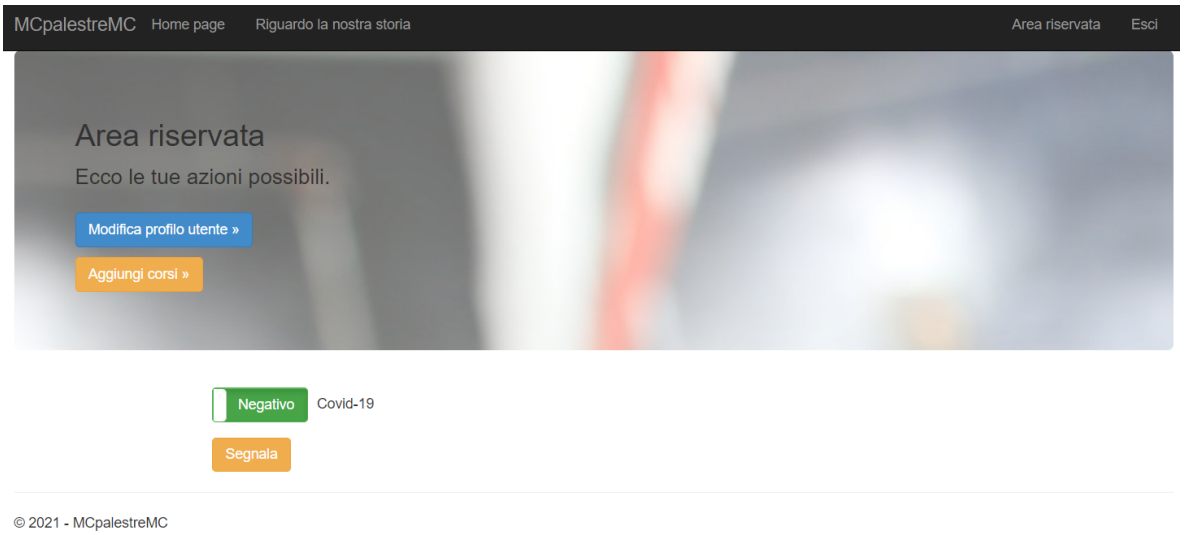
AREA RISERVATA CLIENTE - nuove prenotazioni



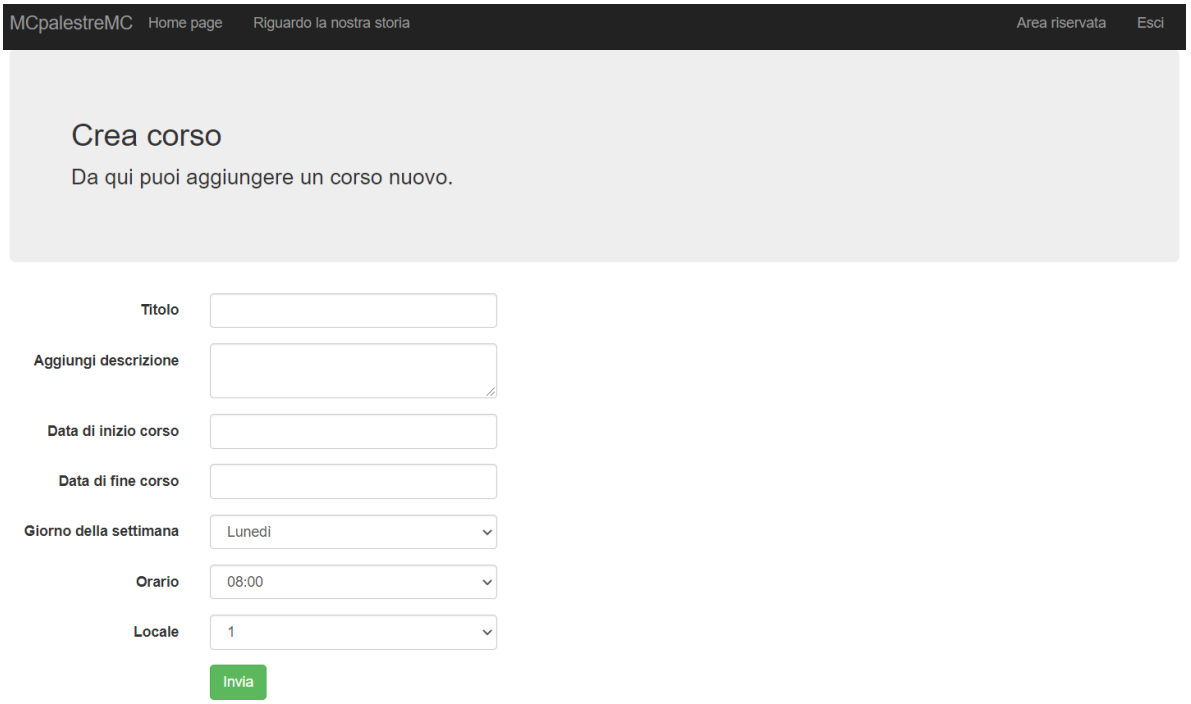
AREA RISERVATA CLIENTE - le tue prenotazioni



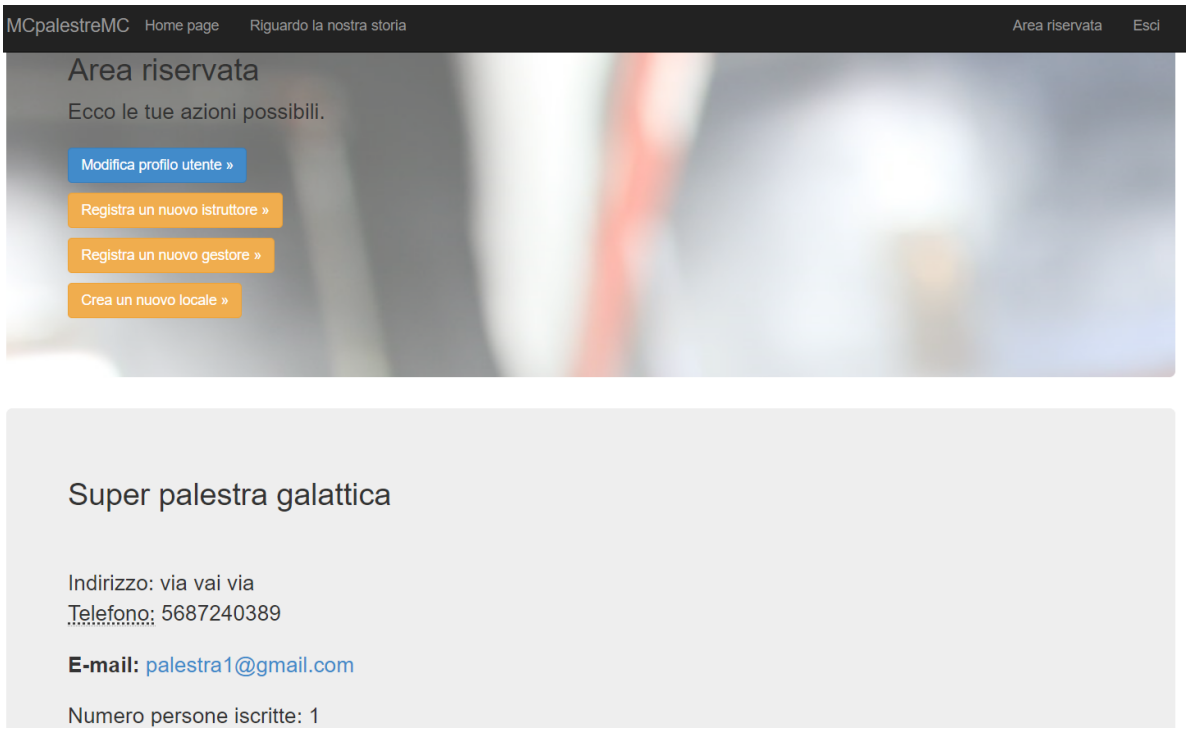
AREA RISERVATA ISTRUTTORE - pagina principale



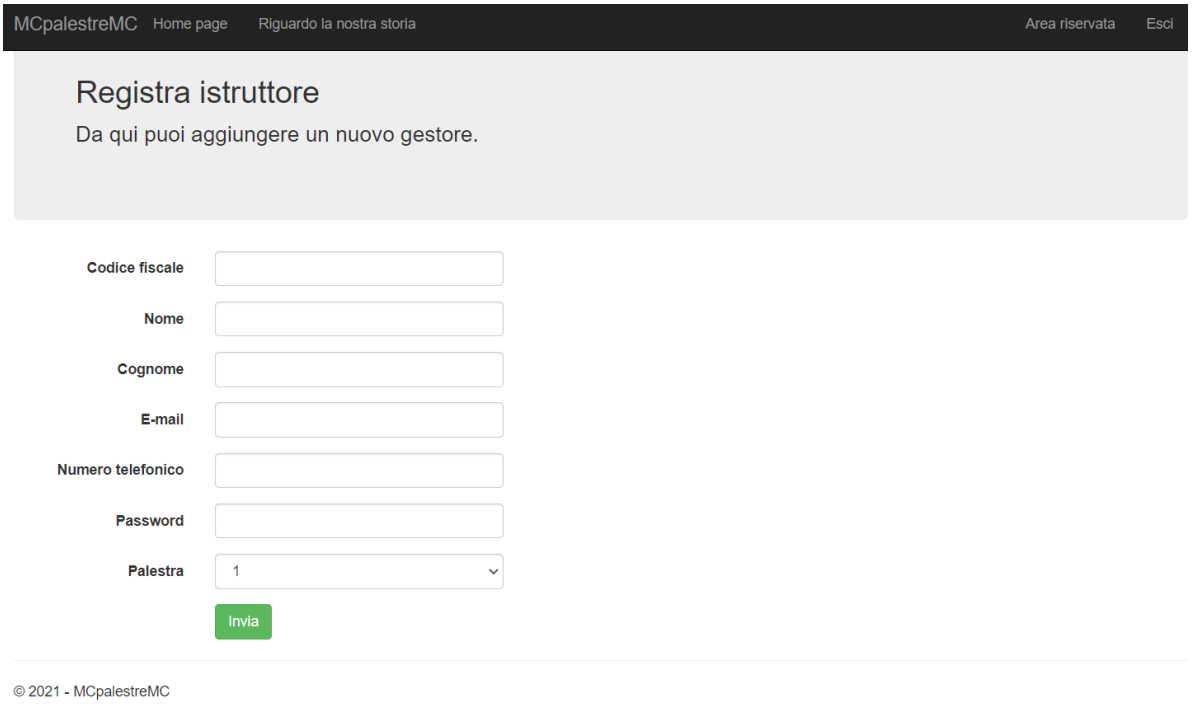
AREA RISERVATA ISTRUTTORE - nuovi corsi



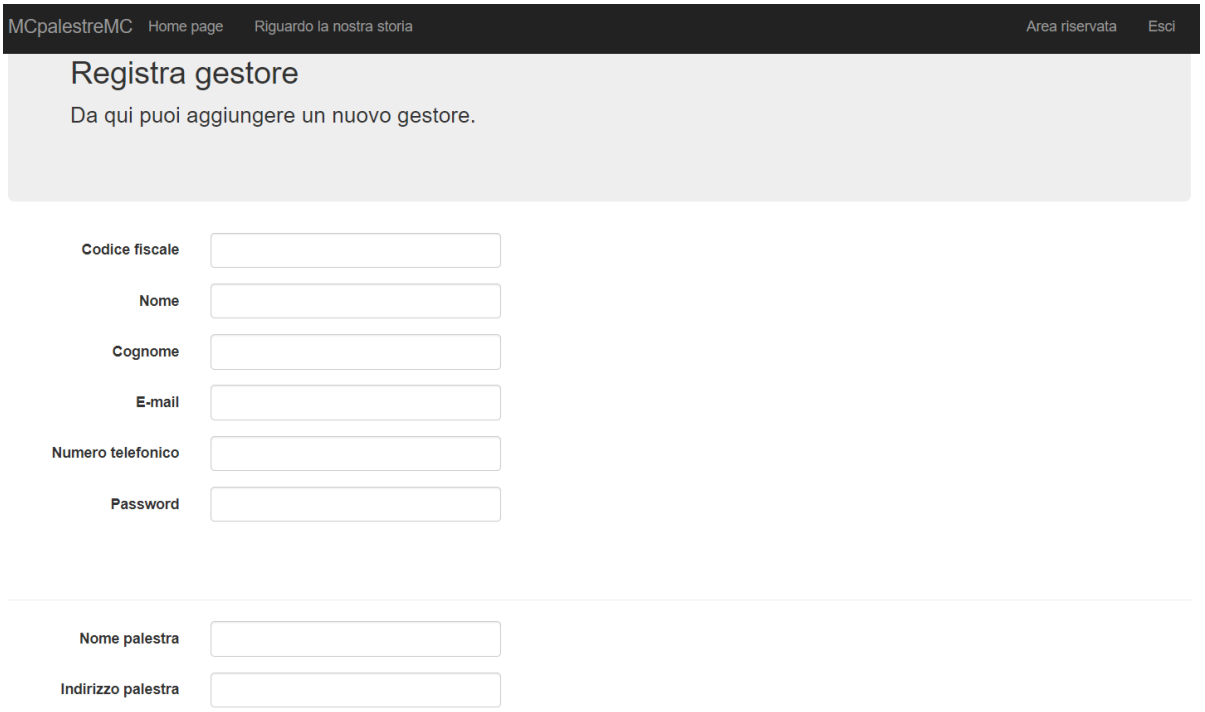
AREA RISERVATA GESTORE - pagina principale

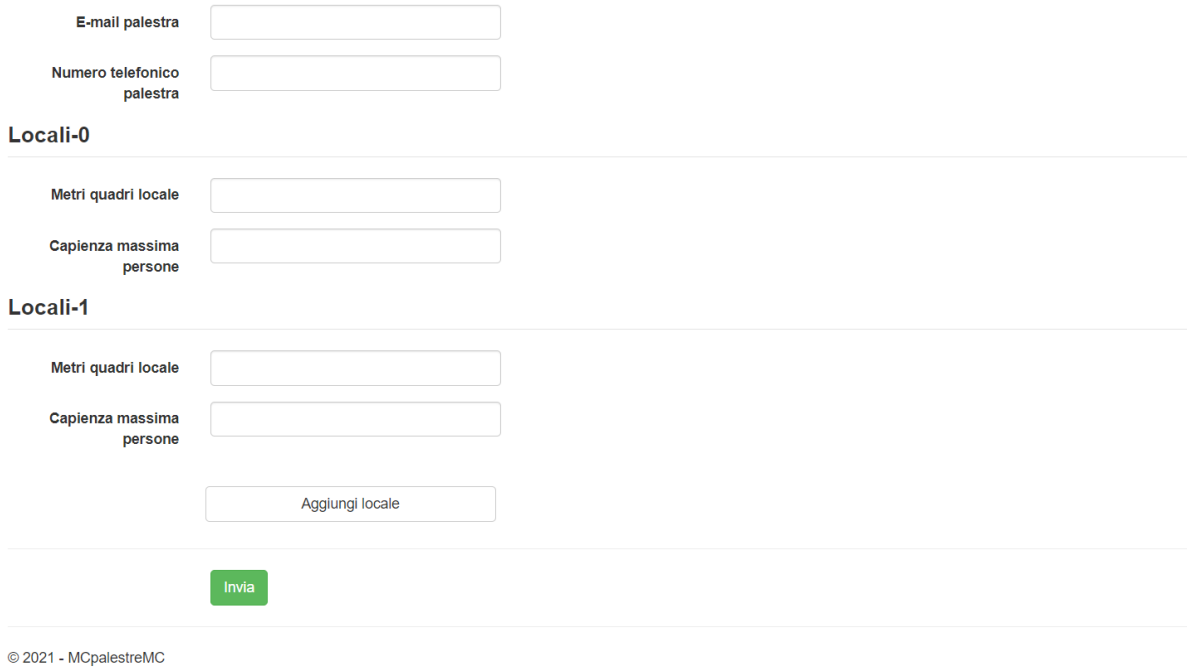


AREA RISERVATA GESTORE - registrazione nuovo istruttore

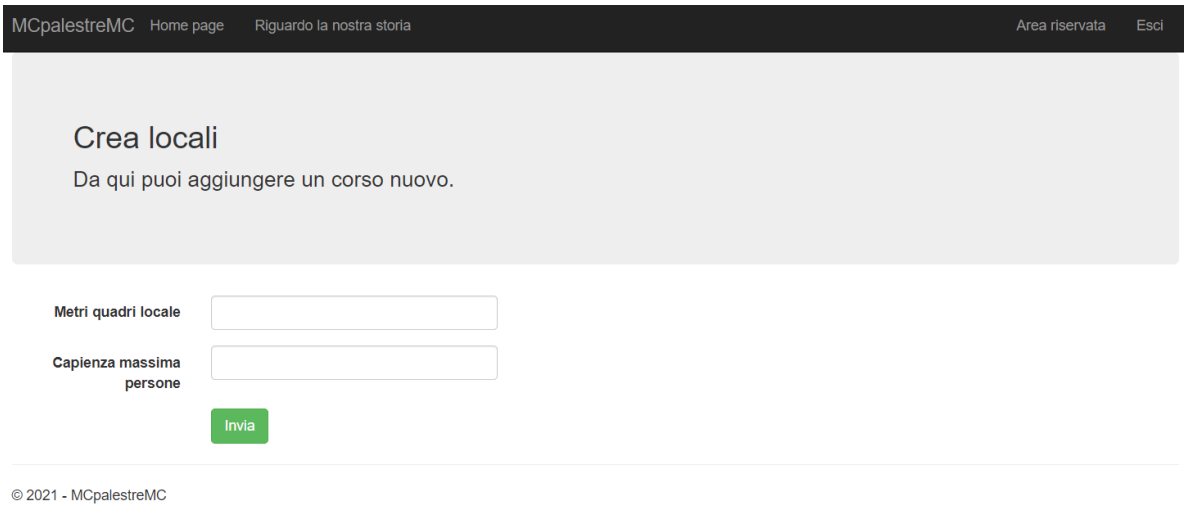


AREA RISERVATA GESTORE - registrazione nuovo gestore + nuova palestra





AREA RISERVATA GESTORE - nuovo locale



4-Progettazione concettuale e logica dello schema della base di dati e i ruoli

Alla luce di quanto detto nei punti precedenti, ora si andrà a presentare lo schema della base di dati con relativa spiegazione delle scelte di progettazione.

Le tabelle in totale sono sei e servono per gestire: utenti, palestre, corsi, prenotazioni, iscrizioni e locali.

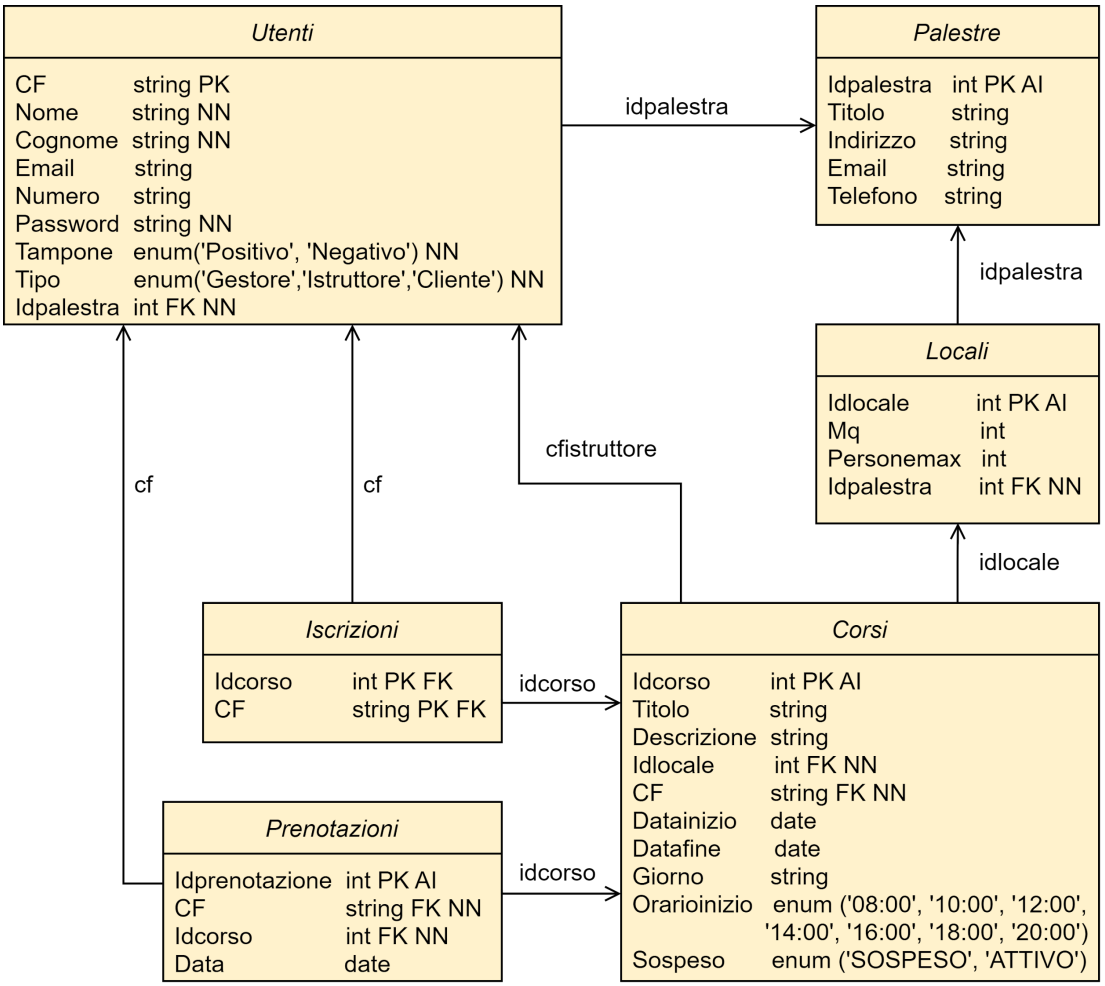
In primis, per gestire gli utenti, c'è un'unica tabella con una colonna "Ruolo" che va ad associare ciascun utente con il proprio ruolo di appartenenza. Questo permette di gestire l'autorizzazione più facilmente all'interno delle funzioni di Flask, ma non va a sostituire i ruoli veri e propri che sono poi assegnati a ciascun utente. I ruoli sono: gestore, istruttore e cliente.

Questa base di dati serve a gestire più palestre, quindi è stata creata una tabella esclusivamente per queste ultime. Questa tabella viene agganciata a quella degli utenti e a quella dei locali, di cui parleremo poi, tramite chiavi esterne presenti in queste altre tabelle che fanno riferimento alla tabella 'Palestre' tramite "idpalestra", la sua chiave primaria. Ogni palestra è gestita da un gestore che, però, non è tra le colonne di questa tabella. Questo perché si sarebbe generata ridondanza. Basta quindi passare per le tabelle corsi e locali per risalire al legame tra ogni palestra ed il proprio gestore. Infatti, ogni utente è collegato ad una sola palestra, sia che sia la sua palestra di proprietà, che ci lavori o che la frequenti.

Per gestire al meglio la capienza, le prenotazioni e le sospensioni dei corsi, si è scelto di optare per separare in diverse tabelle i corsi, i locali nei quali vengono svolti i corsi, le iscrizioni ad un determinato corso e le prenotazioni per un determinato corso ad uno specifico orario e data. Così facendo è possibile gestire separatamente tutti i casi di cui si ha bisogno. Ipotizziamo il caso in cui si ammali un cliente. Lui provvederà a segnalare nella sua area riservata che è positivo al Covid-19 e in automatico (tramite trigger, vedi punto 5.2) i corsi a cui ha partecipato nelle precedenti due settimane verranno sospesi per tutti i clienti con iscrizione a quel corso (impossibilitandoli ad effettuare nuove prenotazioni). Una volta che il cliente torna negativo, dopo aver verificato che nessun altro associato a quello specifico corso sia positivo, il corso ritorna attivo e prenotabile. Lo stesso avviene se si dovesse ammalare l'istruttore. La tabella locali, invece, permette di gestire la capacità massima di persone che possono prenotare un corso in base alla capienza dell'ambiente.

Tutte le tabelle sono interconnesse, ma non ci sono connessioni che potrebbero essere rimosse.

Qui sotto è riportato lo schema completo della base di dati con le varie connessioni.



RUOLI

Parliamo ora di ruoli. I ruoli associati agli utenti sono, come citato sopra, “Cliente”, “Istruttore” e “Gestore”, ma c’è anche un altro ruolo che viene utilizzato in caso di utente anonimo, ovvero “Anonimo”.

Questo ruolo non viene mai associato a nessun utente che abbia eseguito un login, solo a chi accede al sito senza essersi ancora autenticato. L’utente anonimo è l’unico a cui è stato assegnato il ruolo di “Anonimo”.

Gli utenti che hanno il ruolo di “Cliente” sono gli effettivi clienti del sito, quelli che possono effettuare iscrizioni e prenotazioni. Non possono toccare con mano i dati del database, possono solo gestire il proprio profilo, le proprie iscrizioni e le proprie prenotazioni, per il resto, sono come l’utente anonimo.

Gli utenti che hanno il ruolo di “Istruttore” hanno più libertà rispetto ai clienti. Possono gestire il proprio profilo, creare nuovi corsi associandoli ai locali della loro palestra e gestendoli come meglio credono (in base alle disponibilità).

Infine, gli utenti con il ruolo di “Gestore” hanno il vero potere sul database. Possono manipolare la tabella degli utenti, andando ad aggiungere nuovi istruttori o gestori, creare palestre e locali per la propria palestra. Quindi, possono completamente stravolgere la visione che gli istruttori e i clienti hanno dei corsi, dei locali, delle iscrizioni e delle prenotazioni. Tuttavia, i gestori non possono cancellare o cambiare direttamente corsi, iscrizioni e prenotazioni.

5-Scelte progettuali

VINCOLI

In questa sezione si farà riferimento al grafico mostrato al punto 4.

AUTOINCREMENT

Le tabelle palestre, locali, corsi e prenotazioni hanno la primary key settata su autoincrement perché, a differenza, di un utente che può essere identificato tramite un codice (come il codice fiscale), qui bisognerebbe che l'utente decida un codice che non c'è ancora nella tabella interessata. Quindi, dovrebbe avere il permesso di vedere tutta la tabella e andare a verificare manualmente che il codice da lui scelto non sia già presente. Questo non è un buon metodo per scegliere un codice identificativo unico di dati (anche protetti, privati o riservati), meglio optare per una soluzione alternativa, dove il codice viene creato automaticamente.

NOT NULL

Ci sono alcune righe delle tabelle che sono segnate come NN, ovvero Not Null, infatti, quelle righe non ammettono che quel determinato valore non venga inserito. La condizione di primary key ingloba anche la condizione di NN. In ogni caso, per questa web app, non è possibile ottenere una caso in cui uno di questi campi non venga riempito.

PRIMARY KEY

Tutte le tabelle devono presentare un codice identificativo come primary key, ad eccezione della tabella 'Iscrizioni'. Qui si è optato per una doppia primary key che prende in considerazione sia il codice fiscale dell'utente che effettua l'iscrizione, sia il codice identificativo del corso. Entrambe sono sia primary key sia foreign key.

FOREIGN KEY

Il collegamento tra le tabelle avviene tramite foreign key. Il posizionamento delle varie foreign key è banalmente evincibile dal grafico. Non tutte le tabelle hanno foreign key e non verso tutte le altre tabelle. Ogni foreign key presente nel database è fondamentale, non ci sono ridondanze o chiavi superflue.

Ogni foreign key è impostata su 'ON UPDATE CASCADE' e 'ON DELETE CASCADE'.

CHECK

Non ci sono check nè su attributi, nè su tuple.

TRIGGER

Nel progetto sono presenti tre trigger: uno è incentrato sulle modifiche del profilo dell'utente e gli altri due sulla gestione di corsi per la questione Covid-19. Cominciamo proprio da questi.

```
delimiter |
CREATE TRIGGER BloccaCorso AFTER UPDATE ON utenti
    FOR EACH ROW
    BEGIN
        UPDATE corsi c
        SET c.sospeso = 'SOSPESO'
        WHERE c.idcorso IN (SELECT p.idcorso
                            FROM prenotazioni p NATURAL JOIN utenti u
                            WHERE u.tampone = 'Positivo' AND p.data >= CURDATE() - 15
                            AND p.data <= CURDATE()
                            );
    END;
| delimiter ;
```

Il trigger chiamato "BloccaCorso" è stato creato perché nel momento in cui un cliente segnala di essere positivo al Covid-19, i corsi che frequenta devono essere bloccati. Nello specifico, un corso viene settato 'SOSPESO' quando un cliente va a modificare lo status Covid-19 nel suo profilo da 'Negativo' a 'Positivo' e il cliente ha frequentato almeno una lezione nelle precedenti due settimane.

```
delimiter |
CREATE TRIGGER SbloccaCorso AFTER UPDATE ON utenti
    FOR EACH ROW
    BEGIN
        UPDATE corsi c
        SET c.sospeso = 'ATTIVO'
        WHERE c.idcorso IN (SELECT p.idcorso
                            FROM prenotazioni p NATURAL JOIN utenti u
                            WHERE u.tampone = 'Negativo' AND p.idcorso NOT IN(SELECT p2.idcorso
                                                                              FROM prenotazioni p2 NATURAL JOIN utenti u2
                                                                              WHERE u2.tampone = 'Positivo'
                                                                             )
                            );
    END;
| delimiter ;
```

Il trigger chiamato “SbloccaCorso” è stato creato perché nel momento in cui un cliente segnala di essere negativo al Covid-19, i corsi che frequenta devono essere sbloccati (se rispettano alte determinate condizioni). Nello specifico, un corso viene settato ‘ATTIVO’ quando un cliente va a modificare lo status Covid-19 nel suo profilo da ‘Positivo’ a Negativo’ e nessun altro cliente è diventato positivo nel frattempo.

```
delimiter |
CREATE TRIGGER AggiornaProfilo BEFORE UPDATE ON utenti
    FOR EACH ROW
    BEGIN
        IF NEW.Numero = null OR NEW.Numero = '' THEN
            SET NEW.Numero = OLD.Numero;
        END IF;
        IF NEW.Email = null OR NEW.Email = '' THEN
            SET NEW.Email = OLD.Email;
        END IF;
    END;
| delimiter ;
```

Per quanto riguarda il trigger sui dati degli utenti, la questione è molto più semplice. Non si vuole che gli utenti modifichino accidentalmente un dato precedentemente inserito con un niente, cioè un “”. Se nella modifica dei dati del profilo, un utente volesse cambiare solo uno dei campi, non serve che compili anche gli altri e questi non subiranno modifiche. Questo tipo di trigger sfrutta il meccanismo di NEWROW e OLDROW per evitare di sovrascrivere per sbaglio dei dati.

Questo è un esempio di trigger usato per permettere all’utente di non dover compilare tutti i campi modificabili che potrebbe essere sfruttato per tutti gli aggiornamenti dei dati, ma in questo progetto è stato implementato solo in questo caso a scopo esemplificativo.

TRANSAZIONI

La scelta del livello di isolamento per ogni transazione è stata presa in esame singolarmente e poi riesaminata alla luce delle altre scelte prese. Ora, vedremo per ogni caso quale livello di isolamento è stato scelto e perché.

REGISTRAZIONE DI UN CLIENTE - livello: SERIALIZABLE

Questa sezione prevede che ci sia una transazione con livello di isolamento “SERIALIZABLE”. Infatti, se due utenti tentano di registrarsi con lo stesso codice fiscale nello stesso momento, non possono più essere distinti correttamente; quindi serve che avvenga un lock sulla tabella, per permettere ad uno alla volta di accedervi e fare l’inserimento.

REGISTRAZIONE DI UN ISTRUTTORE - livello: SERIALIZABLE

Questa sezione prevede che ci sia una transazione con livello di isolamento “SERIALIZABLE”. Infatti, se due utenti tentano di registrarsi con lo stesso codice fiscale nello stesso momento, non possono più essere distinti correttamente; quindi serve che avvenga un lock sulla tabella, per permettere ad uno alla volta di accedervi e fare l’inserimento.

REGISTRAZIONE DI UN GESTORE CON PALESTRA - livello: SERIALIZABLE

Questa sezione prevede che ci sia una transazione con livello di isolamento “SERIALIZABLE”. Infatti, se due utenti tentano di registrarsi con lo stesso codice fiscale nello stesso momento, non possono più essere distinti correttamente; quindi serve che avvenga un lock sulla tabella, per permettere ad uno alla volta di accedervi e fare l’inserimento.

SEGNALAZIONE COVID-19 - livello: SERIALIZABLE

Questa sezione prevede che ci sia una transazione con livello di isolamento “SERIALIZABLE”. Infatti, se un utente segnala di essere positivo è importante che venga sospeso immediatamente il corso, senza lasciare che altri utenti possano eseguire prenotazioni. Quindi serve che avvenga un lock sulla tabella, per permettere ad uno alla volta di accedervi.

CANCELLAZIONE CORSO - livello: SERIALIZABLE

Questa sezione prevede che ci sia una transazione con livello di isolamento “SERIALIZABLE”. Infatti, se un istruttore decide di cancellare un proprio corso, non deve essere possibile allo stesso momento per un cliente eseguirvi una prenotazione o iscrizione perché sarebbe iscritto o con una prenotazione ad un corso che non esiste più. Quindi serve che avvenga un lock sulla tabella, per permettere ad uno alla volta di accedervi.

CANCELLAZIONE ISCRIZIONE - livello: READ UNCOMMITTED

Questa sezione prevede che ci sia una transazione con livello di isolamento “READ UNCOMMITTED”. Infatti, anche se un cliente cancella i suoi dati d’iscrizione, nessuna altra operazione può essere intaccata.

MODIFICA PROFILO - livello: READ COMMITTED

Questa sezione prevede che ci sia una transazione con livello di isolamento “READ COMMITTED”. Infatti, anche se l’utente modifica i suoi dati, nessun’altra operazione può essere intaccata, ad eccezione di quando si va a modificare la palestra con cui si è associati. In quel caso l’utente deve poter selezionare una palestra inserita definitivamente nel database. Bisogna pertanto aspettare che l’eventuale gestore che sta inserendo una nuova palestra abbia completato l’inserimento.

ISCRIZIONE AD UN CORSO - livello: READ COMMITTED

Questa sezione prevede che ci sia una transazione con livello di isolamento “READ COMMITTED”. Infatti, un cliente deve poter selezionare un corso solo se effettivamente creato da un istruttore. Bisogna pertanto aspettare che l’eventuale istruttore che sta inserendo un nuovo corso ne abbia completato l’inserimento nel database.

CREAZIONE CORSO - livello: SERIALIZABLE

Questa sezione prevede che ci sia una transazione con livello di isolamento “SERIALIZABLE”. Infatti, se due istruttori stanno creando dei corsi nello stesso locale e nello stesso giorno e orario, non devono poter confermare i corsi in sovrapposizione. Quindi serve che avvenga un lock sulla tabella, per permettere ad uno alla volta di accedervi.

PRENOTAZIONE LEZIONE - livello: SERIALIZABLE

Questa sezione prevede che ci sia una transazione con livello di isolamento “SERIALIZABLE”. Infatti, se due clienti stanno contemporaneamente cercando di prenotare l’ultimo posto disponibile per la lezione, non devono poterlo fare perché porterebbe ad un sovraffollamento che non rispetta i limiti di capienza del locale dove si tiene il corso. Quindi serve che avvenga un lock sulla tabella, per permettere ad uno alla volta di accedervi.

CANCELLAZIONE PRENOTAZIONE - livello: READ UNCOMMITTED

Questa sezione prevede che ci sia una transazione con livello di isolamento “READ UNCOMMITTED”. Infatti, anche se un cliente cancella i suoi dati di prenotazione, nessuna altra operazione può essere intaccata.

6-Porzioni di codice interessante e tecnologie specifiche

Vediamo ora un excursus di porzioni di codice interessanti, estrapolate da diverse sezioni del progetto.

SQL

In questa sezione non si andrà ad identificare dove sono ritrovabili questa query, anche per il fatto che alcune sono ripetute (magari con qualche leggera differenza) in più punti.

```
INSERT INTO utenti VALUES(:cf, :nome, :cognome, :email, :numero, :password, 'Negativo', 'Cliente', '1')
create user :codice@'localhost' identified with mysql_native_password by :password
GRANT Cliente to :codice@'localhost'
```

Per registrare un utente, ai prepared statement vengono aggiunti i diversi campi come variabili Flask. Una volta inserito l'utente nella tabella utenti, viene creato anche un user per dargli accesso alla base di dati e gli viene assegnato il ruolo giusto ("Cliente", "Istruttore", "Gestore"). Nel caso del gestore, viene aggiunto "WITH ADMIN OPTION".

```
SELECT p.idpalestra
FROM palestre p
ORDER BY p.idpalestra DESC
LIMIT 1
```

In questa query l'obiettivo era quello di andare a prendere l'ultima entry della tabella. Sappiamo che 'idpaestra' è autoincrementale, quindi notiamo l'uso di ORDER BY DESC per ottenere l'ordine dall'ultima entry alla prima. A questo punto basterà usare LIMIT 1 per andare a prendere solamente la prima selezione che corrisponde all'ultima entry.

```
SELECT p.Titolo, p.Indirizzo, p.Email, p.Telefono, COUNT(u.tipo) AS Personeiscritte
FROM palestre p LEFT JOIN utenti u USING (idpalestra)
WHERE u.tipo = 'Cliente' AND p.idpalestra =:idpalestra
```

In questa query ci sono un paio di particolarità collegate tra di loro. La prima è l'uso di LEFT JOIN nella clausola FROM per ottenere la lista completa della palestre anche senza alcun iscritto. Dato che la tabella 'Palestre' e la tabella 'Utenti' hanno più colonne in comune, qui si usa USING() per specificare la colonna da considerare. L'altra cosa interessante è il COUNT() che va appunto a contare le entry specificate, in questo caso u.tipo. Questo serve per contare quanti iscritti ci

sono per ogni palestra. Se non ci sono iscritti, il conteggio segnerà 0 per quella palestra.

```
SELECT c.Idcorso, c.Titolo, c.Descrizione, c.Idlocale, c.Giorno, c.Orarioinizio, c.Datainizio, c.Datafine, c.Sospeso, COUNT(i.cf) AS Personeiscritte
FROM corsi c LEFT JOIN iscrizioni i ON c.idcorso = i.idcorso
WHERE c.CF =:cf AND c.datafine > CURDATE()
GROUP BY c.idcorso
```

In questa query notiamo l'uso di CURDATE() per risalire alla data attuale e l'uso di GROUP BY per raggruppare gli 'idcorso' in modo da ottenere il risultato desiderato sul COUNT. Infatti, l'obiettivo di questo contatore è conoscere quante persone iscritte ci sono per ogni corso.

```
SELECT c.giorno, c.orarioinizio
FROM corsi c
WHERE c.idlocale =:idlocale AND ((:datainizio BETWEEN c.datainizio AND c.datafine) OR (:datafine BETWEEN c.datainizio AND c.datafine))
```

In questa query molto singolare capiamo il meccanismo nella clausola WHERE. Dopo una prima verifica su 'idlocale', bisognava ricavare tutti i corsi il cui svolgimento si trova, almeno per un giorno, all'interno di una finestra di date delimitate dalla data di inizio e quella di fine che vengono passate. Per farlo, si controlla che o la data di inizio sia compresa tra le date passate con BETWEEN, o la data di fine sia compresa tra le date passate, sempre con BETWEEN.

```
SELECT c.Idcorso, c.Titolo, u.Nome, u.Cognome, c.Descrizione, c.Idlocale, c.Giorno, c.Orarioinizio, c.Datainizio, c.Datafine, c.Sospeso
FROM corsi c NATURAL JOIN utenti u JOIN locali l USING(idlocale)
WHERE l.idpalestra =:idpalestra AND :cf NOT IN (
                                SELECT i.CF FROM iscrizioni i
                                WHERE i.idcorso = c.idcorso
                                )
AND c.datafine > CURDATE()
```

In quest'ultima query ci soffermiamo nuovamente sulla clausola WHERE per far vedere una sottoquery. Nella query principale si ha bisogno di cercare se il 'cf' che viene passato è presente in un'altra selezione. Per fare questo ci si avvale di 'NOT IN' per, appunto, poter aprire un'altra query all'interno della principale da cui si ricaveranno dei risultati tra i quali si farà la ricerca.

FLASK

Passiamo ora ad osservare direttamente il codice scritto in Python Flask.

```python
@app.route('/', methods = ['GET', 'POST'])
@app.route('/home', methods = ['GET', 'POST'])
def home():
    """Renders the home page."""
    return render_template(
        'index.html',
        is_logged = is_logged(),
        title ='Home page',
        year = datetime.now().year,
    )
```

Qui notiamo che quando andiamo a renderizzare la pagina html, passiamo alla funzione anche altri valori, oltre al nome della pagina da renderizzare. Passiamo la variabile ‘is_logged’, calcolata da ‘is_logged()’. Questa funzione restituisce il valore della verifica di autenticazione dell’utente. La variabile ‘is_logged’ serve poi alla pagina html (in realtà a layout.html che index.html estende) per capire come impostare la barra di navigazione.

```python
@app.route('/registrazione', methods=['GET', 'POST'])
def registrazione():
    form = RegistrationForm()
    if form.is_submitted():
        result = request.form
        try:
            cf = result['cf'].upper()
            nome = result['nome']
            cognome = result['cognome']
            email = result['email']
            numero = result['numero']
            password = result['password']

            conn = engine.connect()

            conn.execute("SET TRANSACTION ISOLATION LEVEL SERIALIZABLE")
            conn.execute("START TRANSACTION")

            s = text("SELECT u.CF FROM utenti u WHERE u.CF =:cf")
            id = conn.execute(s, cf = cf).fetchone()

            if id:
                conn.execute("COMMIT")
                conn.close()
                flash('Errore: codice fiscale già registrato','error')
                return redirect(url_for('registrazione'))

            s = text("INSERT INTO utenti VALUES(:cf, :nome, :cognome, :email, :numero, :password, 'Negativo', 'Cliente', '1')")
            rs = conn.execute (s, cf = cf, nome = nome, cognome = cognome, email = email, numero = numero, password = generate_password_hash(password))
            s = text("create user :codice@'localhost' identified with mysql_native_password by :password")
            rs = conn.execute (s, codice = cf, password = password)

            s = text("GRANT Cliente to :codice@'localhost'")
            rs = conn.execute(s, codice = cf)

            rs = conn.execute("FLUSH PRIVILEGES")
            conn.execute("COMMIT")
            conn.close()
            return redirect(url_for('area_riservata'))
        except:
            conn.execute("ROLLBACK")
            conn.close()
            flash("Errore durante la registrazione", 'error')
            return redirect(url_for('registrazione'))
    return render_template(
        "registrazione.html",
        is_logged = is_logged(),
        title = 'Registrazione',
        year = datetime.now().year,
        form = form
        )
```

Prendiamo la funzione di registrazione di un cliente come esempio di diversi aspetti interessanti. Il primo è il riferimento all'engine globale, poi andiamo ad aprire la connessione con il database che verrà chiusa alla fine della funzione. Notiamo l'uso di Textual SQL e Prepared Statement per eseguire le query, ma usiamo text() per salvare il contenuto della variabile 's' e andare poi ad eseguire s. Un'altra cosa interessante è l'uso delle variabili all'interno della query (come abbiamo visto precedentemente): possiamo passare dei valori alle query per andare a confrontare certi elementi con essi, per esempio, nella clausola WHERE. Possiamo anche usare questi valori per inserirlo nel database con un INSERT INTO. Interessante anche l'uso della funzione 'generate_password_hash()' per assicurare un maggiore strato di sicurezza per l'utente. Notiamo anche che, colleghiamo l'utente al database richiamando l'engine. Vediamo l'uso della funzione 'redirect()' che rimanda direttamente ad un'altra funzione per un'altra pagina html. L'ultima cosa da osservare è l'uso di flash per passare messaggi informativi alla pagina html: in questo caso è stato sfruttato per passare messaggi di errore all'utente.

```
434  @app.route('/area_riservata', methods=['GET', 'POST'])
435  @login_required
436  def area_riservata():
437      tipo = current_user.get_tipo()
438      if(tipo == 'Gestore'):
439          return redirect(url_for('area_gestore'))
440      elif(tipo == 'Istruttore'):
441          return redirect(url_for('area_istruttore'))
442      elif(tipo == 'Cliente'):
443          return redirect(url_for('area_cliente'))
444      else:
445         return redirect(url_for('home'))
```

Nella funzione 'area_riservata()' vediamo un meccanismo di autorizzazione dove andiamo a richiamare la funzione 'get_ruolo()' che ricaverà il ruolo dell'utente per reindirizzare ciascun utente verso la propria area riservata. Questo meccanismo è utilizzato in molte funzioni anche per evitare che un utente non autorizzato possa accedere a delle aree del sito a cui non dovrebbe poter accedere. Invece, il meccanismo per controllare l'autenticazione dell'utente risiede nel decoratore '@login_required'. La funzione 'get_ruolo()' è richiamata su 'current_user' per poter ottenere i dati dell'utente corrente. Infatti, la classe 'User' serve per salvare i dati dell'utente corrente.

```
s = text("SELECT c.giorno, c.orarioinizio FROM corsi c WHERE c.idlocale =:idlocale AND ((:datainizio BETWEEN c.data
orari = conn.execute(s, idlocale = idlocale, datainizio = datainizio, datafine = datafine)
lista_orari = []
for o in orari:
    lista_orari.append(o)

selezionato = (giorno, orarioinizio)

if selezionato in lista_orari:
    conn.close()
    flash('Locale già occupato per il giorno e la data selezionati. Scegliere un altro orario o giorno.', 'error')
    return redirect(url_for('crea_corso'))

s = text("INSERT INTO corsi (titolo, descrizione, idlocale, datainizio, datafine, giorno, orarioinizio, cf) VALUES(
conn.execute(s, titolo = titolo, descrizione = descrizione, idlocale = idlocale, datainizio = datainizio, datafine
conn.execute("COMMIT")
conn.close()
return redirect(url_for('area_riservata'))
```

La funzione ‘crea_corso()’ è una delle più interessanti per la meccanica di mappatura di slot e locali disponibili. Quando un istruttore sceglie un locale, un giorno e un orario per il nuovo corso, questo verrà salvato in ‘selezionato’ e dal database verranno selezionati tutti gli altri corsi che appartengono a quello stesso locale e che sono attivi nello slot di giorni selezionato dell’istruttore. Se ‘selezionato’ comparirà nella lista degli orari degli altri corsi, allora verrà rilasciato un messaggio di errore, contrariamente, se non presente sarà uno slot valido per la creazione del corso.

```
result = request.form
corso = result['corso'].split(' ')
idcorso = corso[0]

if result.get("ChooseDate", False):
    conn = engine.connect()

    s = text("SELECT c.datainizio, c.datafine, c.giorno FROM corsi c WHERE c.idcorso =:idcorso ")
    dati_corso = conn.execute(s, idcorso = idcorso).fetchone()

    s = text("SELECT p.data, l.personemax, COUNT(*) AS numeropersone FROM locali l NATURAL JOIN corsi c JOIN prenotazioni
    date_altri = conn.execute(s, idcorso = idcorso)
    s = text("SELECT p.data FROM corsi c JOIN prenotazioni p USING(idcorso) WHERE c.idcorso =:idcorso AND p.cf =:cf")
    date_mie = conn.execute(s, idcorso = idcorso, cf = current_user.get_id())
    conn.close()

    datainizio = dati_corso['datainizio']
    datafine = dati_corso['datafine']
    giorno = dati_corso['giorno']

    lista = []
    for l in date_altri:
        lista.append(str(l['data']))
    for l in date_mie:
        lista.append(str(l['data']))

    date = date_disponibili(str(datainizio), str(datafine), str(giorno))

    date = date_rimanenti(date, lista)

    if len(date) == 0:
        flash('Tutte le date sono già prenotate, scegliere un altro corso','error')
        return redirect(url_for('prenotazioni'))

    form.data.choices = date
```

Un'altra funzione molto interessante è la funzione 'prenotazioni()', soprattutto per quanto riguarda il suo meccanismo per generare la lista di date ancora disponibili da poter prenotare per il corso selezionato. Infatti, vediamo innanzitutto come vengono richiamati i risultati del form tramite 'request.form' e come viene ricavato l' 'idcorso' con la funzione 'split()' sulla funzione che identifica il corso per il cliente. Per la procedura si prosegue in questa maniera: prima viene selezionato il corso scelto dal cliente, poi vengono selezionate tutte le date già piene di prenotazioni al massimo, successivamente vengono selezionate tutte le date già prenotate dal cliente in questione e si uniscono tutte le date ricavate. Per ottenere la lista completa di tutte le date (disponibili o meno) associate al corso selezionato, si richiama la funzione 'date_disponibili()' passando la data di inizio, la data di fine e il giorno della settimana. La funzione provvederà a calcolare tutte le date esatte. Poi, tramite la funzione 'date_rimanenti()' si calcoleranno le date risultati tra le date disponibili e quelle già occupate per avere solo le date con ancora posti liberi. Infine, a meno che non ci siano date libere (in questo caso verrà lanciato un messaggio di errore) verrà settato il campo 'data' del form per la prenotazione con la lista delle date libere. Qui sotto viene riportato il comportamento di queste due funzioni.

```python
def date_disponibili(inizio, fine, giorno):
    date = []
    days_week = ['Lunedì', 'Martedì', 'Mercoledì', 'Giovedì', 'Venerdì', 'Sabato', 'Domenica']
    data = datetime.strptime(inizio, '%Y-%m-%d').date()

    while str(days_week[data.weekday()]) != giorno:
        data = data + timedelta(days = 1)

    while str(data) <= fine :
        date.append(data)
        data = data + timedelta(weeks = 1)

    return date
```

```python
def date_rimanenti(origine, rimozione):
    date = []
    for i in origine:
        date.append(str(i))

    date_finali = []
    for i in date:
        if i not in rimozione:
            date_finali.append(datetime.strptime(i, '%Y-%m-%d').date())

    date_definitive = []

    todayf = datetime.strptime(today, '%Y-%m-%d').date()
    for i in date_finali:
        if i > todayf:
            date_definitive.append(datetime.strptime(str(i), '%Y-%m-%d').date())

    return date_definitive
```

HTML

Qualche punto chiave presente nei file HTML.

```html
<div class="container body-content">
    {% block content %}{% endblock %}
    <hr />
    <footer>
        <p>&copy; {{ year }} - MCpalestreMC</p>
    </footer>
</div>
```

Questo snippet di codice mostra un paio di cose interessanti, entrambe collegate all'uso delle parentesi {}. Vediamo subito la scritta "{% block content %} {% endblock %}" che corrisponde a creare un'apertura nel codice per immettere altri pezzi di codice. Infatti, quando si vorrà andare ad aggiungere del codice per la sezione desiderata, basterà scrivere "{% block xxx%} {% endblock %}", andando a sostituire le xxx con in nome del blocco corrispondente, in questo caso 'content'. La seconda peculiarità è l'uso delle parentesi {{}} doppie. Questo permette di utilizzare variabili passate da Flask, come 'year' in questo caso.

```
1  {% extends "layout.html" %}
2  {% import "bootstrap/wtf.html" as wtf %}
3
4
5  {% block content %}
```

```
19
20 {% endblock %}
```

Qui si vede l'applicazione pratica di ciò che abbiamo detto prima, ovvero, l'apertura del blocco per inserire il 'content'. Si nota anche come prima cosa viene scritto "{% extends "layout.html" %}" per andare a estendere il file html di base 'layout.html' comune a tutte le pagine della web app. Dopodiché viene anche importato il file della libreria di wtforms per poterlo utilizzare nel file html.

```
56      {% with messages = get_flashed_messages(with_categories=true) %}
57      <ul>
58          {% for category, message in messages %}
59          <li class="alert alert-{{ category }} my-2 my-lg-0 ml-5">{{ category }} : {{ message }}</li>
60          {% endfor %}
61      </ul>
62      {% endwith %}
```

In questa sezione di html vediamo la resa dei messaggi informativi passati da Flask con flash. Se ci sono messaggi, questa sezione farà apparire dei messaggi d'errore all'utente. Notiamo anche che la sintassi {%%} permette di utilizzare dei comandi di Flask come i cicli 'for'.

```
33              <div class="navbar-collapse collapse">
34                  <ul class="nav navbar-nav navbar-left">
35                      <li><a href="{{ url_for('home') }}">Home page</a></li>
36                      <li><a href="{{ url_for('about') }}">Riguardo la nostra storia</a></li>
37                  </ul>
38                  <ul class="nav navbar-nav navbar-right">
39                      <li id="nav-areariservata"><a href="{{ url_for('area_riservata') }}">Area riservata</a></li>
40                      <li id="nav-registrazione"> <a href="{{ url_for('registrazione') }}">Registrati</a></li>
41                      <li id="nav-login"><a href="{{ url_for('login') }}">Accedi</a></li>
42                      <li id="nav-logout"><a href="{{ url_for('logout') }}">Esci</a></li>
43                  </ul>
44              </div>
```

In questa sezione di html, si può evincere il meccanismo che gestisce la barra di navigazione che è identica per tutte le schermate. Prima di tutto, si nota l'utilizzo dei comandi Flask tramite le {{}} dove viene appunto richiamata la funzione 'url_for()' per caricare la pagina desiderata. Per quanto riguarda la parte di codice che riguarda l'area riservata, abbiamo già visto come, in base al ruolo, ognuno venga reindirizzato verso l'area riservata corrispondente. La presenza nella barra di navigazione di area riservata, registrazione, login e logout sono invece gestiti da uno script di javascript che vedremo successivamente.

```
{% for c in corsi %}
<div class="btn-grp mx-auto jumbotron mt-5 mb-5">
    <div class="row">
        <div class="col-md-6">
            <h2><b>{{c['Titolo']}}</b></h2>
            <p>
                Istruttore: {{c['Nome']}} {{c['Cognome']}}<br/>
                Locale: {{c['Idlocale']}}<br/>
                Periodo del corso: dal {{c['Datainizio']}} al {{c['Datafine']}}<br/>
                Giorno della settimana: {{c['Giorno']}}<br/>
                Orario: {{c['Orarioinizio']}}<br/><br/>
                {{c['Descrizione']}}
            </p>
        </div>
        {% if c['Sospeso'] == 'SOSPESO' %}
        <div class="col-md-6 sospeso">
            <p>
                Questo corso e' stato sospeso per ragioni Covid-19.
            </p>
        </div>
        {% endif %}
```

Qui si vede un altro comando di Flask, ‘if’, venire usato dentro al file html, ma, cosa più impattante, vediamo come vengono prese le variabili da Flask. Possiamo accedere alle liste e agli array, andando anche a specificarne la chiave dichiarata in precedenza in un file Flask.

JAVASCRIPT

Scorriamo delle brevi sezioni di javascript (jquery) utili al progetto.

```
        $('.alert').each(function (index, element) {
            alert($(element).html());
            this.remove(this);
        });
        if ({{ is_logged }} == true){
            $('#nav-areariservata').removeClass('disabled');
            $('#nav-registrazione').hide();
            $('#nav-login').hide();
            $('#nav-logout').show();
        }
        else{
            $('#nav-areariservata').addClass('disabled');
            $('#nav-registrazione').show();
            $('#nav-login').show();
            $('#nav-logout').hide();
        }
```

Questo pezzo di codice javascript si riferisce alla pagina ‘layout.html’ dove andiamo a manipolare il dom per mostrare o nascondere certi elementi. Come prima cosa, sistemiamo i messaggi che ci arrivano da Flask (che prima abbiamo visto come ottenere nella pagina html) per farli apparire come alert, ovvero popup, e non come scritte nella pagina. Dopodiché possiamo osservare il meccanismo per gestire la presenza nella barra di navigazione di area riservata, registrazione, login e logout di

cui avevamo precedentemente parlato. Tramite la variabile ‘is_logged’, che ci arriva da Flask, andiamo a controllare che l’utente si sia loggato o meno: in caso positivo, l’area riservata verrà sbloccata (abilitata), il bottone di logout verrà mostrato e i bottoni di registrazione e login verranno nascosti; al contrario, in caso negativo, l’area riservata verrà disabilitata, il bottone di logout verrà nascosto e i bottoni di registrazione e login verranno mostrati.

```
65      $(function () {
66          $('#covid').bootstrapToggle({
67              on: 'Positivo',
68              off: 'Negativo',
69              onstyle: 'danger',
70              offstyle: 'success'
71          });
72      });
```

Questo pezzo di codice javascript mostra l’utilizzo del framework Bootstrap per convertire un elemento ‘checkbox’ in un ‘toggle’ con relativa mappatura del significato di ‘on/off’ (che, in quanto checkbox, convergerebbe in ‘checked/unchecked’) e relativo cambiamento di aspetto: rosso per positivo al Codiv-19, verde per negativo al Covid-19.

WTFORMS

Concludiamo col parlare dell’utilizzo della libreria WTForms.

Questa libreria è molto utile per generare in modo rapido e pulito i form per la web app tramite della classi in Pyhton Flask.

```
33  class RoomForm(FlaskForm):
34      mq = FloatField('Metri quadri locale')
35      personeMax = DecimalField('Capienza massima persone')
36
37  class GymManagerRegistrationForm(FlaskForm):
38      cf = StringField('Codice fiscale')
39      nome = StringField('Nome')
40      cognome = StringField('Cognome')
41      email = EmailField('E-mail')
42      numero = StringField('Numero telefonico')
43      password = PasswordField('Password')
44      palestra = StringField('Nome palestra')
45      indirizzo = StringField('Indirizzo palestra')
46      emailPalestra = EmailField('E-mail palestra')
47      telefono = StringField('Numero telefonico palestra')
48      locali = FieldList(FormField(RoomForm), min_entries=2)
49      gymManagerRegistrationSubmit = SubmitField('Invia')
```

Qui vediamo bene un esempio molto chiaro dell’utilizzo di questi form dalla classe ‘GymManagerRegistrationForm’ che presenta una serie di campi eterogenei. Il

campo più classico e comune è ‘StringField’, che rappresenta di per sé una stringa. Un altro campo comune è ‘PasswordField’, che rappresenta appunto una password e che in automatico nella pagina del sito, nasconderà all’occhio dell’utente i caratteri che digiterà. Un campo non base è ‘EmailField’, che rappresenta a sua volta un indirizzo email e che nella pagine si assicurerà autonomamente che in quel campo venga inserita una stringa con formato di una email. Qui abbiamo anche l’occasione di parlare di un campo composto, ovvero ‘FieldList’: questo campo prevede la creazione di una lista di altri tipi di campi, però in questo specifico caso, non andiamo solo ad inserire una lista di campi, ma una lista di form da noi creati. Il form che andiamo a specificare, passa attraverso la dichiarazione ‘FormField’ che stabilisce proprio che si tratta non di un campo qualunque ma di form. A questo punto basterà specificare il nome del form creato precedentemente, in questo caso ‘RoomForm’ e, se si vuole, anche altri parametri, come il numero minimo di entries della lista. Infine, vediamo un ultimo campo comune, 'SubmitField', che rappresenta il bottone di conferma terminazione di compilazione del form. L’ordine con cui si scrivono qui i vari campi, stabilirà l’ordine con cui verranno renderizzati in pagina con la generazione rapida del form che vedremo più avanti.

```
56  class ProfileModificationForm(FlaskForm):
57      email = EmailField('E-mail')
58      numero = StringField('Numero telefonico')
59      idPalestra = SelectField('Palestra', [])
60      profileModificationSubmit = SubmitField('Invia')
61
62      def __init__(self, palestre = None, **kwargs):
63          super().__init__(**kwargs)
64          self['idPalestra'].choices = palestre
```

C’è anche un’altra peculiarità di questi form, ovvero che possiamo andare a popolare dinamicamente il loro contenuto, come nel caso di ‘ProfileModificationForm’. Una volta stabilito il tipo di campo, infatti, possiamo passare alla definizione della classe degli argomenti, calcolati in un altro punto del codice, per settare dinamicamente il loro valore. In questo caso, la lista di ‘idPalestra’ è calcolata dinamicamente sulla tabella ‘Palestre’ e poi passata come argomento.

```
12  <form class="form form-horizontal" action="" method="post">
13      {% for field, errors in form.errors.items() %}
14      {{ ', '.join(errors) }}
15      {% endfor %}
16
17      {{wtf.quick_form(form, form_type="horizontal", button_map={'loginSubmit': 'success'})}}
18  </form>
```

In questo pezzo di codice html, vediamo la corrispettiva implementazione del codice dei form di wtforms lato html. Come prima cosa si crea un tag <form> specificandone il metodo “post”, poi si esegue un controllo nel caso ci fossero errori, infine qui osserviamo la maniera più rapida possibile di impostare l’intero

form. Basta utilizzare la funzione ‘wtf.quick_form()’ e specificare il nome del form (passato come argomento da Flask), il tipo di form e la mappatura dello stile dei bottoni, che in questo caso corrisponde al solo bottone di submit ‘loginSubmit’ che sarà verde.

```
<form class="form form-horizontal" action="" method="post">
    {% for field, errors in form.errors.items() %}
    {{ ', '.join(errors) }}
    {% endfor %}
    {{wtf.form_field(form.corso, form_type="horizontal")}}
    <input id="add-new-field" class="btn btn-default col-lg-offset-2" type="submit" name="ChooseDate" value="Scegli data"/>
    <br/><br/><hr/>
    {{wtf.form_field(form.data, form_type="horizontal")}}
    {{wtf.form_field(form.bookingSubmit, form_type="horizontal", button_map={'bookingSubmit': 'success'})}}
</form>
```

Adesso vediamo una piccola variante rispetto all’implementazione precedente dei wtforms, in quanto non si renderizza l’intero form con una sola funzione, ma si sfrutta un’altra funzione, ‘wtf.form_field()’, per generare manualmente ogni campo. Questo è utile quando, come in questo caso, in base alla scelta dell’utente, si deve andare a generare dinamicamente il contenuto di un campo. Per farlo, si inserisce un tag <input> che fungerà da invio al codice Flask per entrare nel ramo ‘if’ che genererà la lista di date, come abbiamo visto precedentemente. Dopodiché basterà continuare a specificare manualmente gli altri campi del form, compreso il bottone finale con tanto di mappatura dello stile.

```
<form class="form form-horizontal" action="" method="post">
    {% for field, errors in form.errors.items() %}
    {{ ', '.join(errors) }}
    {% endfor %}

    {{wtf.form_field(form.cf, form_type="horizontal")}}
    {{wtf.form_field(form.nome, form_type="horizontal")}}
    {{wtf.form_field(form.cognome, form_type="horizontal")}}
    {{wtf.form_field(form.email, form_type="horizontal")}}
    {{wtf.form_field(form.numero, form_type="horizontal")}}
    {{wtf.form_field(form.password, form_type="horizontal")}}

    <br/><br/><hr/>

    {{wtf.form_field(form.palestra, form_type="horizontal")}}
    {{wtf.form_field(form.indirizzo, form_type="horizontal")}}
    {{wtf.form_field(form.emailPalestra, form_type="horizontal")}}
    {{wtf.form_field(form.telefono, form_type="horizontal")}}

    <div id="field-list-forms">
        {% for field in form.locali %}
        {{wtf.form_field(field, form_type="horizontal")}}
        {% endfor %}
        <br/>
        <input id="add-new-field" class="btn btn-default col-lg-offset-2 col-lg-10" type="submit" name="AddField" value="Aggiungi lo
        <br/><br/><hr/>
    </div>

    {{wtf.form_field(form.gymManagerRegistrationSubmit, form_type="horizontal", button_map={'gymManagerRegistrationSubmit': 'success
</form>
```

Questo ultimo esempio è molto interessante per mostrare l’implementazione dell’aggiunta dinamica dei ‘FormField’ che abbiamo visto prima. Avevamo

specificato un numero minimo di entries, ma il numero effettivo si può cambiare a seconda di ciò che l'utente vuole ottenere. Lo possiamo gestire semplicemente tramite un tag al cui interno si va a ciclare sul campo lista interessato (dato che non se ne conosce il numero esatto) per renderizzarli singolarmente. Con un tag <input>, l'utente può specificare di aggiungere una nuova entry alla lista e tramite Flask questa verrà effettivamente aggiunta e renderizzata. Chiaramente l'utente può aggiungere quante entries desidera.

7-Futuro della web app

Ci sono degli aspetti che sicuramente si possono sviluppare maggiormente in questo progetto e che, se dovesse essere utilizzato per una vera compagnia di palestre, sarebbero significativi.

La prima cosa sarebbe quella di aggiungere la possibilità per gli utenti di fare delle ricerche mirate per mostrargli i corsi che più si abbinano al loro stile sportivo: come risultante si potrebbe avere un sito che garantisce un introito più alto.

Un'altro oggetto che si potrebbe prendere in considerazione è quello dei 'Veri abbonamenti': fornire al cliente dell'app una serie di iscrizioni con relativi pagamenti online che per ovvietà non sono stati implementati.

Un'ulteriore operazione che dovrebbe essere implementata dovrebbe essere quella della cancellazione(non sviluppata per questioni di praticità tecnica): l'utente è destinato per sempre ad avere un abbonamento ad MCpalestreMC. Meno male che sono gratuite.

Queste sono alcune tra le aggiunte che porterebbero la web app ad un livello più alto, ma ci sono altre proposte che non ho preso in considerazione che potrebbero essere applicate.