
PROGETTO -SISTEMI APERTI E DISTRIBUITI



Tasso Tennis - prenotazione campi da tennis

STUDENTI:

Riommi Maria [315912]

Vescera Nicolò [301838]



Obiettivi

Ideare e realizzare la progettazione di un Web Service che permetta di gestire in modo interattivo la prenotazione, da parte di un utente Player, e l'inserimento, da parte di un utente Latifondista, di campi da tennis usando le tecnologie SOAP/REST su una piattaforma a scelta e un database per l'archiviazione dei dati.

Tecnologie utilizzate

- Python + Flask
- REST/Json
- Bootstrap
- JQuery + AJAX
- database SQLite

Realizzazione

Funzionalità comuni a tutti gli utenti

- **Homepage iniziale:** pagina visualizzata al primo collegamento.
- **About:** pagina con informazioni riguardanti la Tasso Tennis.
- **Registrazione:** pagina per registrare un nuovo utente.
- **Login:** pagina per l'autenticazione degli utenti.
- **Visualizzazione del profilo:** presente nella dashboard dell'utente, contiene dati dell'utente e un button per il logout.

Login

Email

Password

Login

Form di Login

Join Today

Username

Name

Surname

Email

Password

Confirm Password

☐ Landowner?

Sign Up

Form di Registrazione

maria

Welcome back maria !

Name: Maria Riommi

User: maria

Email: maria@maria.com

Type: LandOwner

Logout

Visualizzazione dati utente autenticato

Descrizione tipologia di utenti del servizio

A seguito della registrazione si distinguono due tipi di attori (tramite una checkbox presente nell'apposito form) :









- **Latifondista:** utente che possiede dei campi da tennis e decide di inserirli nella piattaforma al fine di permetterne la prenotazione da parte di utenti Player.
- **Player:** utente che usufruisce del Web Service per prenotare i campi da tennis disponibili per un limitato lasso di tempo.

Funzionalità per utente Latifondista

- **Inserimento, modifica ed eliminazione di campi da tennis:** al login l'utente viene reindirizzato alla dashboard contenente la lista di tutti i campi da tennis che gli appartengono, con la possibilità di eliminazione e modifica. Per l'inserimento di un nuovo campo è presente un button che rimanda alla pagina contenente il form da compilare con tutte le informazioni (nome, descrizione, indirizzo, orari di apertura e chiusura del campo e prezzo per ora)

Dashboard

I tuoi campi:

Nome	Descrizione	€/ora	Indirizzo		
Campo1	Descrizione sintetica del Campo 1	€12.0	via delle macchie 33		
Campo2	Descrizione sintetica del Campo 2	€33.0	via arcobaleno 420		
Campo3	Descrizione sintetica del Campo 3	€9.0	ramazzano (le pulci)		
Campo4	Descrizione sintetica del Campo 4	€1.0	via cortonese 11		

Aggiungi un campo

- **Visualizzazione prenotazioni effettuate dai Player:** sempre nella dashboard vengono visualizzate tutte le prenotazioni effettuate dagli utenti Player relative ai campi posseduti dall'utente.

Le prenotazioni sui tuoi campi:

Data	Inizio	Fine	Campo	Prezzo	Utente
2021-06-13	09:00	11:00	Campo1	€ 24.0	lello
2021-06-22	02:00	04:00	Campo3	€ 18.0	lello

Funzionalità per utente Player

- **Visualizzazione ed eliminazione delle prenotazioni effettuate:** al login l'utente viene reindirizzato alla dashboard contenente la lista di tutte le prenotazioni effettuate con la possibilità di eliminazione.

Dashboard

Le tue prenotazioni:

Data	Inizio	Fine	Campo	Prezzo	
2021-06-13	09:00	11:00	Campo1	€ 24.0	
2021-06-22	02:00	04:00	Campo3	€ 18.0	
2021-06-16	11:00	12:00	Campo4	€ 1.0	

[Effettua una nuova prenotazione](#)

- **Prenotazione di campi da tennis:** tramite l'apposito bottone l'utente verrà reindirizzato alla pagina per effettuare le prenotazioni contenente tutti i campi disponibili. Cliccando sul campo desiderato verrà visualizzato un modal al cui interno è presente il form da compilare per completare la prenotazione.

Tasso Tennis Prenotazioni Home About

Prenota

Tutti i campi disponibili:

Nome	Descrizione
Campo1	Descrizione sintetica del
Campo2	Descrizione sintetica del
maria	Maria
Campo3	Descrizione sintetica del
Campo4	Descrizione sintetica del

Prenota il campo

Nome: Campo1

Descrizione: Descrizione sintetica del Campo 1

Indirizzo: via delle macchie 33

Apertura: 08:00 **Chiusura:** 20:00

€/ora: 12.0

Da: --:-- **A:** --:-- **Giorno:** gg / mm / aaaa

Chiudi Prenota

lo

come back lello !

Name: lollo marchi

User: lello

Email: lello.kawasaki@libero.it

Type: User

Logout

Registrazione

La logica della registrazione viene gestita lato server che controlla che i dati necessari siano tutti inseriti. Successivamente controlla che non siano già presenti utenti con lo stesso username e la stessa email nel database. Nel caso non vengano trovate corrispondenze viene generata l'hash della password e il nuovo utente viene creato.

```
@app.route('/register', methods=["GET", "POST"])
def register():
    # controlla se l'utente e' gia' autenticato
    if current_user.is_authenticated:
        return redirect(url_for('dashboard')) # reindirige alla pagina dashboard

    form = RegistrationForm() # prepara il form di registrazione

    if form.validate_on_submit():
        # prende i dati del form
        user = form.username.data
        pwd = form.password.data
        name = form.name.data
        surname = form.surname.data
        email = form.email.data
        landowner = form.landowner.data
        confirm_pwd = form.confirm_password.data

        # vedo se esistono utenti con la stessa email o stesso username
        existing_user = User.query.filter_by(username=user).first()
        existing_email = User.query.filter_by(email=email).first()

        if not existing_user and not existing_email:
            if pwd == confirm_pwd: # controlla che la password sia giusta
                hashed_password = bcrypt.generate_password_hash(pwd)
                new_user = User(
                    username=user,
                    password=hashed_password,
                    name=name,
                    surname=surname,
                    email=email,
                    landowner=landowner
                ) # creo il nuovo utente

                # aggiungo l'utente al database
                db.session.add(new_user)
                db.session.commit()

                # crea il messaggio da stampare
                flash(f'Account created for {form.username.data}!', 'success')
                return redirect(url_for('home'))
            else:
                flash(f'Password and Confirm Password doesn\'t match !', 'danger')
        else:
            if existing_user:
                flash(f'Username {form.username.data} already in use !', 'danger')

            if existing_email:
                flash(f'Email {form.email.data} already in use !', 'danger')

    return render_template('registration.html', title="Register", form=form)
```

Autenticazione

Come per la registrazione anche l'autenticazione viene gestita lato server che controlla se esiste un utente nel database con l'email corrispondente all'email inserita nell'apposito form e nel caso controlla che le hash delle password corrispondano.

```
@app.route('/login', methods=["GET", "POST"])
def login():
    form = LoginForm() # prepara il form di login

    # controlla se l'utente e' gia' autenticato
    if current_user.is_authenticated:
        return redirect(url_for('dashboard')) # redirige alla pagina dashboard

    if form.validate_on_submit():
        # prende i dati del form
        email = form.email.data
        pwd = form.password.data

        user = User.query.filter_by(email=email).first() # cerca l'utente con la data email
        if user:
            # se trova l'utente controlla la password
            if bcrypt.check_password_hash(user.password, pwd):
                login_user(user) # effettua il login
                flash('You have been logged in!', 'success')

                return redirect(url_for('dashboard'))

            flash('Login Unsuccessful. Please check username and password', 'danger')

        return render_template('login.html', title="Login", form=form)
```

API

Convenzioni utilizzate

In base al tipo di metodo con cui vengono chiamate le API si distinguono due diverse convenzioni sul valore di ritorno:

- **GET:**

in caso di successo viene ritornata una stringa json formattata come segue:

```
{'rep': [dati_richiesti]}
```

In caso di errore la stringa json sarà: `{'error': 'Messaggio di errore'}`

- **POST, PUT, DELETE:**

in caso di successo viene ritornata una stringa json formattata come segue:

```
{'message': 'Messaggio di successo'}
```

In caso di errore la stringa json sarà: {'error': 'Messaggio di errore'}

/api/fields

- **GET:**

In base alla tipologia dell'utente vengono distinti due comportamenti:

- **Latifondista:**

Ritorna solo i campi posseduti dall'utente.

- **Player:**

Ritorna tutti i campi presenti nel database.

```
@app.route('/api/fields', methods=['GET'])
@login_required
def get_all_fields():
    # controlla che l'utente sia autenticato
    if current_user.is_authenticated:
        if current_user.landowner:
            user_id = current_user.id

            # prende tutti i campi
            dati = Field.query.filter_by(landowner_id=user_id)

            # converte il risultato in dizionario per poter essere inviato a JS
            dati_json = [x.as_dict() for x in dati]

            return json.dumps({'rep': dati_json}), 200 # ritorna i dati

        else:
            dati = Field.query.all() # prende tutti i campi

            # converte il risultato in dizionario per poter essere inviato a JS
            dati_json = [x.as_dict() for x in dati]

            return json.dumps({'rep': dati_json}), 200

    return json.dumps({'error': 'Login first !'}), 401
```

- POST:

Controlla se l'utente autenticato è di tipo latifondista e nel caso tenta l'inserimento nel database di un nuovo campo.

```
@app.route('/api/fields', methods=['POST'])
@login_required
def add_field():
    if current_user.is_authenticated and current_user.landowner:
        user_id = current_user.id

        # prende i valori passatigli
        name = request.form['name']
        description = request.form['description']
        address = request.form['address']
        available_from = datetime.strptime(request.form['available_from'], '%H:%M').time()
        available_to = datetime.strptime(request.form['available_to'], '%H:%M').time()
        price_h = request.form['price_h']

        # creo il nuovo Field da aggiungere al db
        new_field = Field(
            name=name,
            description=description,
            address=address,
            available_from=available_from,
            available_to=available_to,
            price_h=price_h,
            landowner_id=user_id
        )

        # prova ad aggiungere il campo
        try:
            db.session.add(new_field)
            db.session.commit()
        except:
            return json.dumps({'error': 'Esiste gia\' un campo con questo nome !'}), 400

        return json.dumps({'message': f'Campo {name} aggiunto con successo !'}), 200

    return json.dumps({'error': 'Login first !'}), 401
```

- DELETE:

Elimina dal database il campo selezionato permettendo l'operazione solo se l'utente autenticato è di tipo latifondista e se possiede il campo. Verranno eliminate a cascata anche tutte le prenotazioni relative al campo.

```
@app.route('/api/fields/<int:id>', methods=['DELETE'])
@login_required
def delete_field(id):
    if current_user.is_authenticated and current_user.landowner:
        user_id = current_user.id

        # prova a trovare il campo da eliminare
        to_delete = Field.query.filter_by(id=id, landowner_id=user_id).first()
        if to_delete:                                # se viene trovato lo elimina
            name = to_delete.name
            field_id = to_delete.id
            db.session.delete(to_delete)
            db.session.commit()

            # elimina in cascata anche le prenotazioni relative a quel campo
            prenotazioni_to_delete = Prenotation
                                .query
                                .filter_by(field_id=field_id)
                                .all()

            for elem in prenotazioni_to_delete:
                db.session.delete(elem)
                db.session.commit()

            return json.dumps({'message': f'Campo {name} eliminato con successo !'}), 200

        return json.dumps({'error': 'Campo inesistente !'}), 400

    return json.dumps({'error': 'Login first !'}), 401
```

- **PUT:**

Modifica nel database le informazioni (nome, descrizione e indirizzo) del campo selezionato dall'utente permettendo l'operazione solo se l'utente autenticato è di tipo latifondista e se possiede il campo.

```
@app.route('/api/fields', methods=['PUT'])
@login_required
def update_field():
    # controlla che l'utente sia un latifondista
    if current_user.is_authenticated and current_user.landowner:
        # prende i dati dal form
        user_id      = current_user.id
        name          = request.form['name']
        description   = request.form['description']
        address       = request.form['address']
        field_id      = request.form['field_id']

        # controlla che il campo che viene modificato appartiene all'utente
        to_update = Field.query.filter_by(id=field_id, landowner_id=user_id).first()
        if to_update:
            to_update.name      = name
            to_update.description = description
            to_update.address   = address

            # prova a modificare il campo
            try:
                db.session.commit()
            except:
                return json.dumps({'error': 'Nome del campo gia\' esistente !'}), 400

            return json.dumps({'message': f'Campo {name} aggiornato con successo !'}), 200

        return json.dumps({'error': 'Campo inesistente !'}), 400

    return json.dumps({'error': 'Login first !'}), 401
```

/api/prenotations

- **GET:**

In base alla tipologia dell'utente vengono distinti due comportamenti:

- **Latifondista:**

Ritorna tutte le prenotazioni effettuate dagli utenti Player per i propri campi.

- **Player:**

Ritorna tutte le prenotazioni effettuate dall'utente.

```
@app.route('/api/prenotations', methods=['GET'])
@login_required
def get_all_prenotations():
    # controlla che l'utente sia autenticato
    if current_user.is_authenticated:
        # distingue le due tipologie di utenti
        if current_user.landowner == False:
            user_id = current_user.id

        # effettua la join tra Prenotazione e Campo per ottenere il nome del campo
        prenotazioni = db.session.query(
            Prenotation.date,
            Prenotation.start,
            Prenotation.end,
            Prenotation.price,
            Field.name,
            Prenotation.id
        ).join(
            Field,
            Prenotation.field_id == Field.id
        ).filter(Prenotation.player_id == user_id)

        # converte le prenotazioni in JSON per essere ritornate
        dati_json = []
        for elem in prenotazioni:
            tmp = {
                'date': str(elem[0]),
                'start': str(elem[1])[:-3],
                'end': str(elem[2])[:-3],
                'price': str(round(elem[3], 1)),
                'name': str(elem[4]),
                'id': str(elem[5])
            }

            dati_json.append(tmp)

        return json.dumps({'rep': dati_json}), 200
```

```
else:
    user_id = current_user.id

    # effettua la join tra Prenotazione e Campo per ottenere il nome del campo
    prenotazioni = db.session.query(
        Prenotation.date,
        Prenotation.start,
        Prenotation.end,
        Prenotation.price,
        Field.name,
        Prenotation.id,
        User.username
    ).join(
        Field,
        Prenotation.field_id == Field.id
    ).join(
        User,
        Prenotation.player_id == User.id
    ).filter(Field.landowner_id == user_id)

    # converte le prenotazioni in JSON per essere ritornate
    dati_json = []
    for elem in prenotazioni:
        tmp = {
            'date': str(elem[0]),
            'start': str(elem[1])[:-3],
            'end': str(elem[2])[:-3],
            'price': str(round(elem[3], 1)),
            'name': str(elem[4]),
            'id': str(elem[5]),
            'username': str(elem[6])
        }

        dati_json.append(tmp)

    return json.dumps({'rep': dati_json}), 200

return json.dumps({'error': 'Login first !'}), 401
```

- POST:

Controlla se l'utente autenticato è di tipo Player e nel caso tenta l'inserimento nel database di una nuova prenotazione controllando che i dati siano validi:

- Controlla che la data non sia antecedente alla data del giorno in cui viene effettuata la prenotazione.
- Controlla che l'ora della prenotazione rispetti gli orari di apertura e chiusura del campo.
- Controlla che la prenotazione non si sovrapponga alla altre.

```
@app.route('/api/prenotations', methods=['POST'])
@login_required
def add_prenotation():
    # controlla che l'utente sia un utente normale
    if current_user.is_authenticated and current_user.landowner == False:
        user_id = current_user.id

        # prende i dati per la nuova prenotazione
        field_id = request.form['field_id']
        start = datetime.strptime(request.form['da'], '%H:%M')
        end = datetime.strptime(request.form['a'], '%H:%M')
        date = datetime.strptime(request.form['giorno'], '%Y-%m-%d').date()

        price_h = Field.query.filter_by(id=field_id).first().price_h
        price = price_h * diff_time_time(start, end) # calcola il prezzo totale

        # crea la nuova prenotazione
        new_prenotation = Prenotation(
            field_id=field_id,
            player_id=user_id,
            date=date,
            start=start.time(),
            end=end.time(),
            price=price)

        # controlla se la prenotazione e' valida
        is_valid = validate_prenotation(new_prenotation)

        if is_valid[0] == False:
            return json.dumps(
                {'error': f'Impossibile effettuare la prenotazione: {is_valid[1]}'},
                400)

        # prova ad aggiungere la prenotazione al db
        try:
            db.session.add(new_prenotation)
            db.session.commit()
        except:
            return json.dumps({'error': 'Impossibile effettuare la prenotazione!'}), 400

        # se arriva fino qui vuol dire che la prenotazione e' stata aggiunta al db
        return json.dumps({'message': 'Prenotazione effettuata con successo !'}), 200

    return json.dumps({'error': 'Login first !'}), 401
```

Funzione di validazione della prenotazione:

```
def validate_prenotazione(prenotazione):
    today = datetime.today().date() # prende la data di oggi

    # controlla che la data di prenotazione non sia passata
    if prenotazione.date >= today:
        # prende tutte le prenotazioni fatte in quella data
        prenotazioni_same_date = Prenotation.query.filter_by(date=prenotazione.date,
field_id=prenotazione.field_id)

        s = prenotazione.start # ora di inizio prenotazione
        e = prenotazione.end # ora di fine prenotazione

        # controlla che l'ora di inizio e fine non siano invertite
        if s > e:
            return False, 'L\'ora non e\' corretta !'

        # controllo che la prenotazione sia di almeno 1 ora
        ss = datetime.strptime(str(s), '%H:%M:%S')
        ee = datetime.strptime(str(e), '%H:%M:%S')

        if diff_time_time(ss, ee) < 1.0:
            return False, 'Prenota almeno 1 ora !'

        # controlla che l'ora di prenotazione rispetti gli orari del campo
        campo = Field.query.filter_by(id=prenotazione.field_id).first()
        if s >= campo.available_from and e <= campo.available_to:
            pass
        else:
            return False, 'La tua prenotazione non rispetta gli orari del campo !'

        # controlla che la prenotazione non si sovrapponga con altre
        for elem in prenotazioni_same_date:
            i = elem.start
            j = elem.end

            if s < i and e <= i:
                pass
            elif s >= j and e > s:
                pass
            else:
                return False, 'Il campo e\' prenotato per quell\'ora !'

        # se arriva fino qui vuol dire che la prenotazione e' valida !!
        return True, ''
    else:
        return False, 'Controlla la data !'
```

- **DELETE:**

Elimina dal database la prenotazione selezionata permettendo l'operazione solo se l'utente autenticato è di tipo Player e se ha effettuato la prenotazione.

```
@app.route('/api/prenotations/<int:id>', methods=['DELETE'])
@login_required
def delete_prenotation(id):
    # controlla che l'utente sia un utente normale
    if current_user.is_authenticated and current_user.isowner == False:
        user_id = current_user.id

        # elimina la prenotazione
        to_delete = Prenotation.query.filter_by(id=id, player_id=user_id).first()
        if to_delete:
            db.session.delete(to_delete)
            db.session.commit()

            return json.dumps({'message': f'Prenotazione eliminata con successo !'}), 200

        return json.dumps({'error': 'Prenotazione inesistente !'}), 400

    return json.dumps({'error': 'Login first !'}), 401
```

Richieste AJAX

Lato client le API vengono utilizzate tramite richieste AJAX, di seguito andremo ad elencare una chiamata per tipologia di metodo usato:

- **GET:**

la seguente chiamata prende i dati di tutte le prenotazioni effettuate sui campi dell'utente e le stampa a video sotto forma di tabella.

```
$.ajax({
  url: '/api/prenotations',
  type: 'GET',

  success: function(response) {
    risposta = JSON.parse(response);    // converte la risposta in JSON
    dati = risposta.rep;                // prende i dati ritornati

    // crea e popola la tabella
    var tabella = $("
```

- **POST:**

La seguente chiamata invoca l'API per l'inserimento di un nuovo campo.

```
$.ajax({
  url: '/api/fields',
  data: $('form').serialize(),    // prende i dati del form
  type: 'POST',

  success: function(response) {
    risposta = JSON.parse(response);    // converte la risposta in JSON

    alert(risposta.message);            // stampa il messaggio di successo
    $('form').trigger('reset');         // svuota il form
  },

  error: function(error) {
    errore = JSON.parse(error.responseText); // converte la risposta in JSON
    alert(errore.error);                  // stampa l'errore
  }
});
```

- **DELETE:**

La seguente chiamata invoca l'API per l'eliminazione del campo.

```
$.ajax({
  url: '/api/fields/'+id,
  type: 'DELETE',

  success: function(response) {
    risposta = JSON.parse(response);    // converte la risposta in JSON
    alert(risposta.message);            // avvisa dell'avvenuto successo
    location.reload();                  // aggiorna la pagina
  },

  error: function(error) {
    errore = JSON.parse(error.responseText)    // converte la risposta in JSON
    alert(errore.error);                      // avvisa dell'errore
  }
});
```

- **PUT:**

La seguente chiamata invoca l'API per la modifica dei dati del campo.

```
$.ajax({
  url: '/api/fields',
  data: $('#modifica_form').serialize(),    // prende i dati del form
  type: 'PUT',

  success: function(response) {
    risposta = JSON.parse(response);    // converte la risposta in JSON

    // scrive il messaggio sul box per gli avvisi
    $('#errori').text(risposta.message).attr('class', 'alert alert-success');
  },

  error: function(error) {
    errore = JSON.parse(error.responseText);    // converte la risposta in JSON

    // scrive l'errore sul box per gli avvisi
    $('#errori').text(errore.error).attr('class', 'alert alert-danger');
  }
});
```

Struttura del Database

Per la gestione del Database abbiamo utilizzato la libreria SQLAlchemy che permette la definizione di tabelle, query, etc... in Python in modo indipendente dalla tipologia di database utilizzata.

Il database realizzato per la gestione del Web Service è strutturato in tre tabelle:

- **User:** tabella contenente i dati degli utenti. Le password vengono salvate sotto forma di hash.

```
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(30), nullable=False, unique=True)
    name = db.Column(db.String(30), nullable=False)
    surname = db.Column(db.String(40), nullable=False)
    email = db.Column(db.String(40), unique=True, nullable=False)
    password = db.Column(db.String(20), nullable=False)
    landowner = db.Column(db.Boolean(), default=False)
```

- **Field:** tabella contenente i dati dei campi inseriti dai latifondisti.

```
class Field(db.Model):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.String(20), nullable=False, unique=True)
    description = db.Column(db.Text)
    address = db.Column(db.String(60), nullable=False)
    available_from = db.Column(db.Time, nullable=False)
    available_to = db.Column(db.Time, nullable=False)
    price_h = db.Column(db.Float, nullable=False)
    landowner_id = db.Column(db.String(30), db.ForeignKey('user.id'), default=False)
```

- **Prenotation:** tabella contenente i dati delle prenotazioni effettuate dagli utenti player.

```
class Prenotation(db.Model):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    field_id = db.Column(db.Integer, db.ForeignKey('field.id', ondelete='CASCADE'))
    player_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    date = db.Column(db.Date, nullable=False)
    start = db.Column(db.Time, nullable=False)
    end = db.Column(db.Time, nullable=False)
    price = db.Column(db.Float, nullable=False)
    __table_args__ = (db.UniqueConstraint(
        'field_id', 'date', 'start', 'end', name='_prenotation_uc'),
    )
```