Gestione campo tiro con l’arco

di Nocetti Gabriele Matricola N. 145532

## Traccia del progetto

Il progetto nasce dall’esigenza, specialmente dopo l’esperienza con la pandemia, di aiutare la gestione di un campo di tiro con l’arco in diversi aspetti. La traccia concordata prevede funzionalità in base al tipo di utente che visita il sito, prima di tutto è necessario quindi elencare i tipi di utente previsti.

#### Allievi

Utenti registrati, limitati nella fase di prenotazione dove potranno occupare il campo solo se nella stessa ora è presente almeno un altro utente di tipo *Maestro*.

#### Standard

Utenti registrati, pieno accesso a tutte le fasce orarie possibili.

#### Maestri

Utenti registrati, senza limitazioni, una volta effettuata una prenotazione per una certa ora, permettono ai tipo *Allievi* di prenotare per la stessa ora.

#### Anonimi

Utenti non registrati, possono solamente visualizzare il sito in modalità “solo lettura” e chiedere informazioni al gestore tramite una chat real-time.

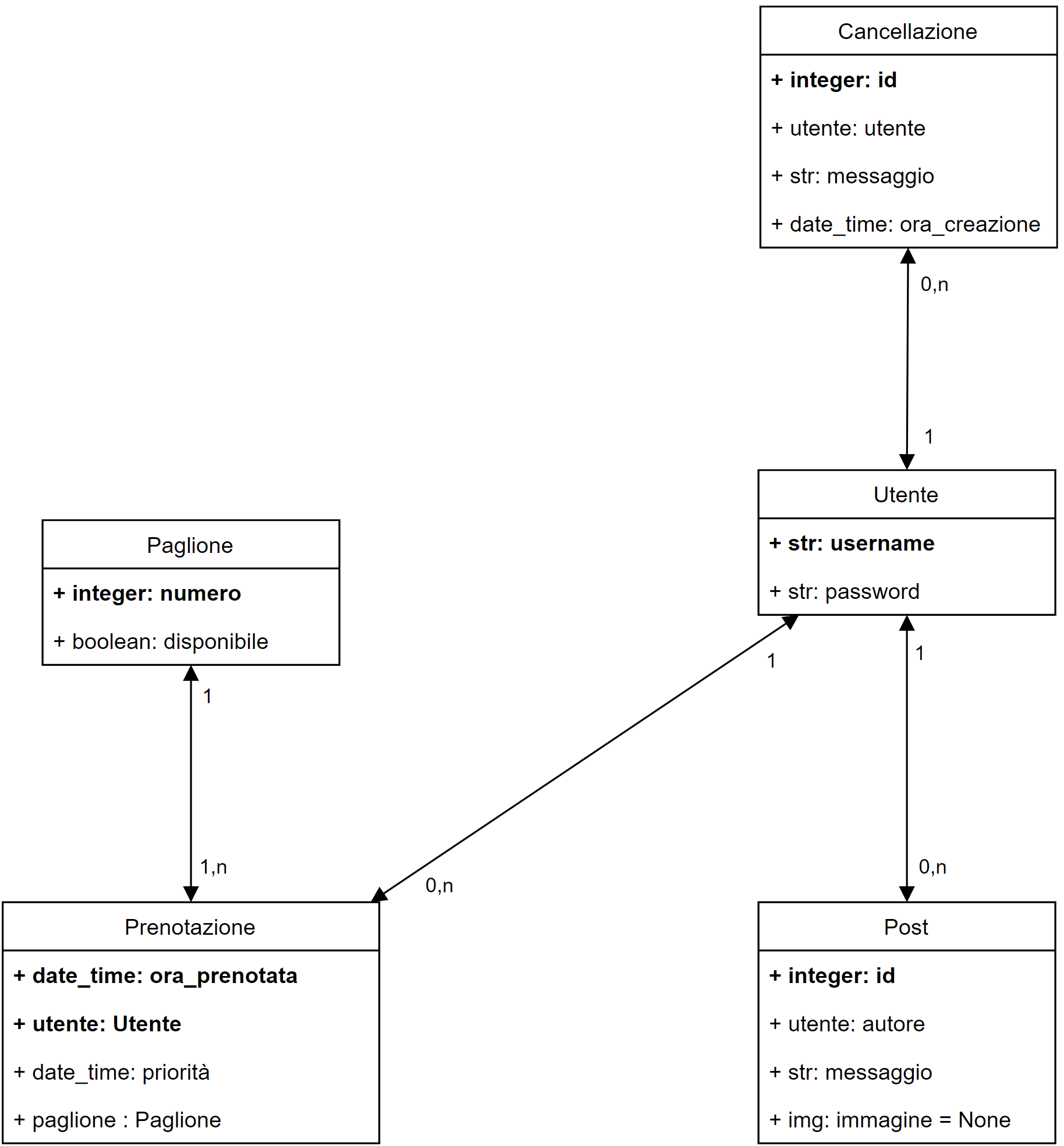
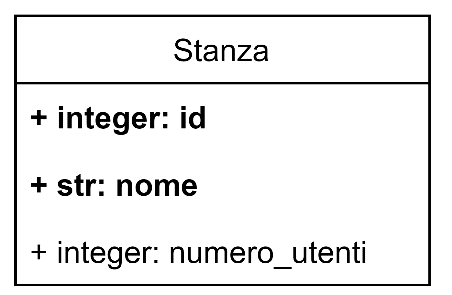
Risulta evidente che la funzionalità principale è la prenotazione e la personalizzazione della schermata in base alle possibilità di cui l’utente dispone. A queste prenotazioni si aggiunge una lista di attesa e nel caso di cancellazione, l’utente in coda verrà notificato. Sarà presente anche una bacheca, dove ogni utente registrato potrà postare messaggi ed immagini legate al campo come, comunicazioni di tornei, frecce perse e strumenti ritrovati.

Dalla piattaforma dell’amministratore si potrà elevare o abbassare il grado di un utente seguendo questo ordine, *Allievi*<->*Standard*<->*Maestri*, perciò un *Allievo* prima di essere *Maestro* dovrà passare al tipo *Standard* e viceversa.

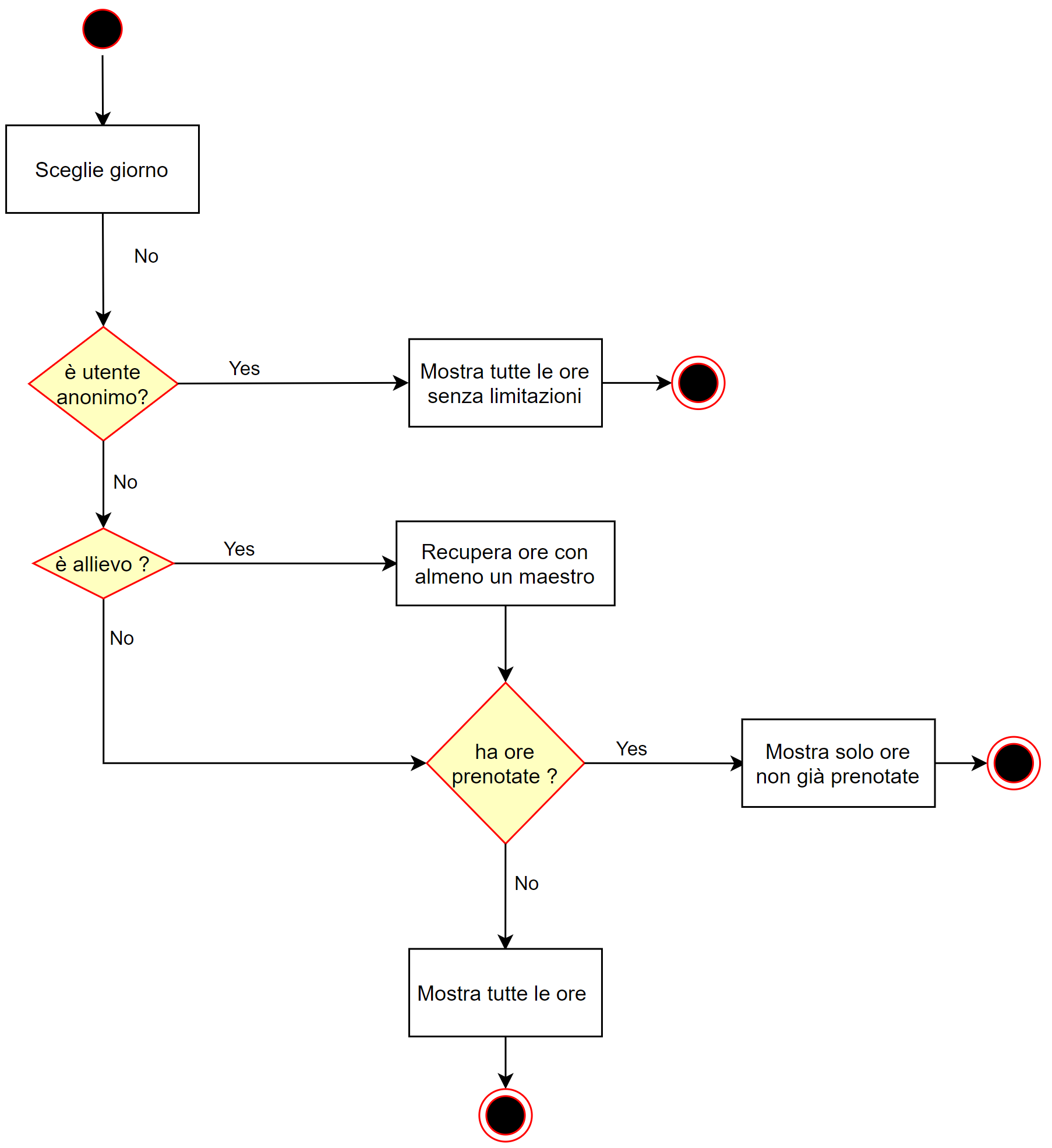
## Descrizione del progetto

Il campo sarà composto da dieci paglioni, ognuno di essi potrà essere attivo o no tramite un flag modificabile solo dall’amministratore. La prenotazione, con chiave primaria composta da utente ed ora\_prenotata, imporrà ad ogni utente di avere al massimo una prenotazione per ogni fascia oraria. Ogni post avrà il suo autore, un Utente, un messaggio ed un’immagine opzionale. Per gestire le notifiche di cancellazione la tabella Cancellazione avrà il messaggio, l’utente al quale è diretta e l’ora in cui è stata creata. Ultima tabella è “Stanza”, non collegata a nessun’altra perché si occupa di gestire la lista di chat anonime attualmente in corso.

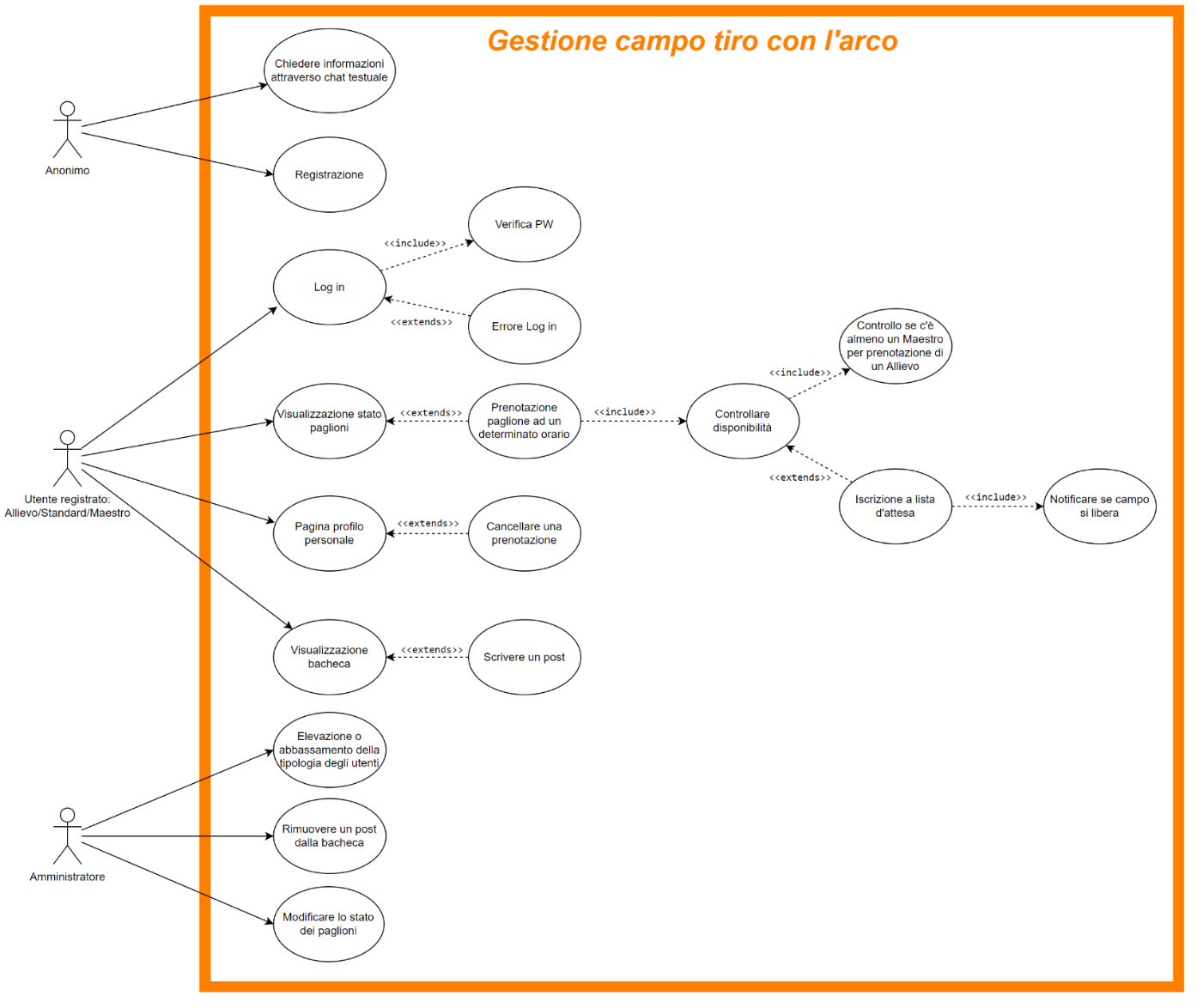
### Diagramma UML



### Activity Diagram della fase di prenotazione



### Diagramma Use Case



## Tecnologie usate

SQLite, oltre che ad essere usata di default da Django, usata avere maggiore controllo e soprattutto eseguire codice in sintassi SQL richiamando la console del database con il comando:

python manage.py dbshell

Django Channels, progetto che include numerosi packages che introducono la gestione di web-socket e supporto per codice async direttamente su Python. Verranno utilizzati per realizzare il servizio di chat.

JavaScript, linguaggio di programmazione utilizzato con script dentro pagine HTML, indispensabile per mostrare in tempo reale i messaggi scambiati da una chat real-time.

CSS, linguaggio per descrivere lo stile e la formattazione di pagine HTML, risulta di grande aiuto quando si vuole offrire all’utente una schermata chiara e di semplice utilizzo.

GitHub, servizio di hosting in cloud di repository Git, offre version control ed altri strumenti aggiuntivi, usato principalmente come diario di aggiornamento per tenere traccia di ogni funzionalità implementata.

Pipenv, strumento ottenibile come package Python, è un gestore di pacchetti ed ambienti virtuali per Python, usato per facilitare l’ottenimento delle dipendenze del progetto e per tenerle in ordine.

## Organizzazione logica dell’applicazione

Una volta chiarita ed approfondita la traccia con i diagrammi UML, Activity ed Use Case si è passati alla creazione del progetto Django *campo*. La suddivisione in app segue le principali funzionalità da implementare ed in tutto saranno cinque. Amministratore, bacheca, messaggi, prenotazioni ed utenti.

Partendo dall’app utenti si sono usati i modelli User, Group, le schermate di registrazione, login e logout offerte da Django. L’unica modifica è stata effettuata nel form di registrazione, dove si setta il nuovo utente di default appartenente solo al gruppo *Allievi*. All’interno è presente anche la view dei dettagli dell’utente contenente informazioni personali come le prenotazioni effettuate e le possibili notifiche.

L’app prenotazioni, la più importante, contiene i primi modelli custom del progetto. Paglione, Prenotazione e Cancellazione sono infatti inseriti nel file /models.py. Django vieta di avere all’interno di un suo modello più di un campo con il flag primary\_key = True, di conseguenza per descrivere la tabella Prenotazione correttamente è stato sovrascritto il campo constraint della meta-classe. Sono state create numerose view e per evitare ridondanza di codice è stata utile la funzione \_lista\_ore\_prenotabili, usata per gestire le ore disponibili nel caso di utente di tipo *Allievo* o no. Infine, è stato aggiunto il file /initcmds.py contenente una configurazione solamente iniziale che crea dieci paglioni e casualmente imposta il campo attivo.

La configurazione dell’app Bacheca ha riguardato la creazione del modello Post e successivamente le pagine di creazione e visualizzazione, in questo caso differenziandosi dall’app precedente perché si è fatto uso di sole CBV. Creazione necessaria è stata quella di una cartella nella root del progetto, /media/images per garantire il salvataggio e la visualizzazione delle immagini inserite tramite form dagli utenti nei Post.

L’app messaggi contiene tutto ciò che permette alla chat real-time di funzionare, dal file /consumer.py, /routing.py alle view necessarie. Il modello descritto Stanza ha come unico utilizzo quello di ordinare le chat attualmente attive per poi mostrarle all’amministratore. Anche in questo caso è presente il file /initcmds.py che tramite thread cancella periodicamente le stanze vuote.

Si può notare come l’app amministratore non risulti tra INSTALLED\_APPS in /campo/settings.py, questo è motivato da una scelta compiuta durante lo sviluppo del progetto. Per garantire una modularizzazione più chiara contrariamente alla divisione iniziale in 5 app, è stato deciso di inserire i vincoli e le view dell’amministratore all’interno delle stesse app. In questo modo, per ogni app, nel file /admin.py sono presenti i vincoli di modifica, scrittura e visualizzazione di ogni modello. Discorso analogo per le view, protette da un test is\_admin, sono accessibili solamente se l’utente è l’admin e sono presenti nelle app utenti e messaggi, la prima per gestire i gruppi degli utenti e la seconda per visualizzare ed interagire con le chat. La gestione del flag attivo di ogni paglione è stata relegata al pannello Django con le dovute limitazioni descritte in /prenotazioni/admin.py, discorso analogo per l’eliminazione di post dalla bacheca.

## Scelte fatte sulle funzionalità implementate

Inizialmente la pagina di prenotazione era un semplice form, faticoso da compilare e che permetteva all’utente di selezionare orari non validi prima di confermare. Per risolvere questa operazione macchinosa, è stato scelto un approccio che può anche richiedere più click da parte dell’utente, ma garantisce la restituzione di sole prenotazioni che l’utente può effettivamente prenotare. I primi controlli avvengono dopo la selezione del giorno (i prossimi sette giorni partendo da oggi) e restituisce le ore, in base al gruppo di appartenenza di chi ha effettuato il login, in cui l’utente non è già occupato. Oltre a bloccare eventuali accessi tramite URL a date non comprese nel range dei prossimi sette giorni. Selezionata l’ora corretta, all’utente viene restituita una lista di paglioni attivi, e per ogni paglione una lista di attesa.

Per implementare questa lista si è pensato a registrare nel campo priorità del modello Prenotazione l’ora esatta in cui tale prenotazione viene creata, ed attraverso una query ordinata per priorità si risale alla lista di attesa del paglione. Di grande utilità è stato il metodo primo\_priorità che restituisce True solo se la prenotazione interrogata è in cima a questa lista.

All’utente di tipo Allievo è stata imposta la possibilità di effettuare una prenotazione solo a paglioni liberi e solo se un maestro è primo nella lista di attesa di un altro paglione. Una volta creata la prenotazione, verrà cancellata nel caso in cui il maestro cancelli la prenotazione, imponendo all’allievo l’accesso al campo solo se supervisionato da un maestro. Queste operazioni di cancellazione sono state protette richiedendo il campo utente della prenotazione uguale all’utente che chiama la cancellazione.

Una volta cancellata la prenotazione, se c’è qualcun altro in coda, sarà necessario notificarlo della liberazione del paglione, e per ottenere questo si è decorata la funzione di creazione notifica tramite un @receiver con campo pre\_delete dalla libreria django.db.models.signals. Inserita in prenotazione/signals.py, la funzione viene chiamata ogni volta che si cancella un’entità di tipo Prenotazione e se rispetta tutti i vincoli elencati qui sopra, salverà nella tabella Cancellazioni una nuova cancellazione che sarà visualizzabile dalla pagina profilo dell’utente in questione. Di stessa concezione è stata scritta la funzione elimina\_prenotazioni\_non\_attivo, la quale se un paglione passa dallo stato attivo True -> False, elimina tutte le prenotazioni legate ad esso.

Per lo sviluppo della chat real-time si volevano ottenere i seguenti risultati, creazione di una chat isolata per ogni utente anonimo che la richiede, con possibilità data solo all’amministratore di chattare con gli utenti. Riguardo la creazione, una volta aperta la pagina di chat, all’utente è chiesto l’inserimento di una sola parola “oggetto” della domanda. Questa stringa diventerà il nome della stanza e verrà aggiunta all’url del web socket per isolarla da eventuali stanze di diverso nome. Un dato così sensibile come questa stringa ha bisogno di controlli, ecco perché all’interno dello script di JavaScript si verifica oltre che la stringa non sia vuota, l’assenza di spazi e di caratteri che potrebbero creare problemi. Nel /consumers.py è stato gestito il limite di accesso a massimo due utenti a chat, la creazione ed anche l’aggiornamento dei campi delle istanze di tipo Stanza. A questo punto l’utente anonimo ha creato una stanza e nella vista da amministratore StanzaView sarà comparso un link per accedere alla chat. Tramite il thread periodico, descritto nella fase di organizzazione logica, le chat vuote (senza utenti) verranno cancellate. Il limite più grande, riscontrato con questa soluzione, è il caso in cui un utente, scrivendo l’oggetto della domanda (room\_name) identico ad un’altra chat attualmente attiva, abbia accesso a quella stanza che diventerà popolata da due utenti anonimi invece che da un anonimo ed un amministratore. Tale limite potrebbe essere risolto in futuro, con la creazione di due classi consumer di tipo diverso, differenziando l’accesso per tipo di utente.

## Testing

La fase di testing ha interessato l’app prenotazioni, iniziando dall’esaminazione di tutti i casi possibili della funzione elimina\_prenotazioni\_paglione\_non\_attivo. Nel setup vengono creati gruppi, utenti, paglioni e prenotazioni di prova. Primo test è il caso base, paglione attivo con prenotazioni salvate viene disattivato. Si verifica la cancellazione, quindi l’esistenza di nessuna prenotazione associata. Verificato anche il contrario, il campo attivo del paglione passando da False a True non cancella nessuna prenotazione, si possono iniziare i test riguardanti i tipi di utente. Un caso possibile è la disabilitazione di un paglione prenotato da un maestro, il quale ha permesso ad un utente allievo la prenotazione. Andrà testata la corretta eliminazione della prenotazione associata all’allievo e la non cancellazione di tutte le altre prenotazioni collegate ad utenti standard o maestro.

Successivamente si è effettuato un test completo, con uso di test client, sulla view GiornoView. Il funzionamento di tale view prevede un parametro richiesto day, un controllo di formattazione della data passata tramite il parametro, la verifica che il giorno scelto sia presente nella lista dei prossimi sette a partire da oggi ed i controlli per tipo di utente rappresentatati nell’activity diagram. Superato il setup, che risulta simile al precedente, si può iniziare a testare la vista senza registrazione. Il primo caso riguarda un utente anonimo che accede alla pagina /giorno/non%20sono%20una%20data/orari/, ci si aspetta perciò il reindirizzamento al settimo giorno a partire da oggi ed una vista senza limitazioni per quanto riguarda gli orari. Prossimo test, ancora utente anonimo ma visitando la pagina del giorno di domani, oltre alla verifica della libertà di visualizzare gli orari, viene aggiunto un controllo al titolo della schermata, il quale deve coincidere il giorno richiesto (domani). L’utilizzo del metodo force\_login(user) del client, permette il testing della view di un utente registrato, senza dimenticare il logout() a fine test. Il tipo allievi è il primo esaminato e non avendo attualmente prenotazioni di maestri si vuole verificare l’assenza di orari disponibili per questo tipo. Si passa al tipo standard, con una prenotazione attiva, si controlla l’assenza di orari in cui è già impegnato e la presenza di ore in cui è libero da prenotare. Con l’ultimo utente, il tipo maestro senza prenotazioni, si esegue un solo controllo sulla presenza di orari disponibili. Il prossimo test, eliminando la prenotazione dell’utente standard, conferma ora la visualizzazione dell’ora appena liberata da parte dell’utente. Il test finale riguarda l’associazione tra allievo e maestro, creata una prenotazione di un maestro, si esegue l’accesso come allievi e si verifica la corretta presenza nella pagina della fascia oraria in comune con il maestro.

I test, manipolando giorni, ore e date, richiamano al loro avvio una serie di warning legati ad un conflitto di compatibilità tra il modulo datetime (usato nel test) ed il formato usato da Django per salvare questi dati. Questi warning causati da incongruenze riguardanti i fusi orari (timezone), non compaiono in fase di utilizzo del sito, poiché quando si ottiene il contenuto di campi DateTime del database, si imposta il campo tzinfo=None.

## Problemi riscontrati

Oltre a quelli già presentati, in fase di scrittura del codice molte idee sono state implementate in modo differente da quello pensato inizialmente, per motivi di complicatezza e di effettiva progettazione. Il più grande ha riguardato le notifiche delle prenotazioni libere. L’idea iniziale prevedeva un sistema in tempo reale, utilizzando ajax oppure anche con channels come per la chat, che periodicamente controllava lo stato della tabella prenotazioni ed in caso di cambiamenti notificava l’utente. Questo sistema era poco efficiente e richiedeva che l’utente stesse visualizzando la sua schermata personale per ottenere una notifica. Documentandosi su un’alternativa più leggera e semplice, si è deciso di sacrificare il real-time, utilizzando funzioni decorate da @receiver, che eseguono solo quando indicate dalle condizioni dal campo signals sulla tabella sender.

Successivamente, un limite da definire era l’appartenenza ad un utente di un solo gruppo e per arrivare alla soluzione attuata è necessaria una premessa. Django permette la creazione di utenti custom ed una best practice è farlo all’inizio del progetto, quando ancora non sono state costruite view e modelli sulla base del tipo utente. Nel momento in cui si voglia aggiungere/modificare la classe User e/o Group a progetto iniziato, come nel mio caso, sono quindi necessari numerosi passaggi anche macchinosi, di migrazioni, manipolazione di database ed altro poiché per come è stato sviluppato Django quest’operazione è consigliabile farla all’inizio. Avendo seguito diverse guide il problema che si riscontrava più spesso era il seguente:

ERRORS: <class 'utenti.admin.UserAdmin'>: (admin.E020) The value of 'filter\_horizontal[0]' must be a many-to-many field.

Analizzata la documentazione si è confermato che l’errore è collegato alla volontà di impostare l’associazione Utente-Gruppo come uno a molti, causando un problema di compatibilità con la logica di Django. Riflettuto su questo, la soluzione trovata ha come idea la sostituzione della sezione modifica della tabella User e Group nel pannello da amministratore Django con la creazione di una pagina ad hoc per ciò. Tale pagina è protetta da possibili accessi di chi non sia amministratore e si presenta come una lista di tutti gli utenti e la possibilità per ognuno di elevare o abbassare il grado/gruppo come presentato nella traccia iniziale.

Schermata iniziale utente vs admin

Passi per prenotazione

Cancellazione con notifica

Schermate admin personalizzate

Chat

### Fare alla fine di tutto gli screenshot

### Home del sito



### Home con accesso da amministratore

Immagine che contiene testo

Descrizione generata automaticamente