cirullino bot

Un bot di telegram per giocare a cirulla con gli amici.

Questo bot è scritto in JavaScript ed utilizza il framework Telegraf.js. Questo framework è molto capace ma ha una documentazione molto povera e mal strutturata. Avrei preferito scrivere il codice in TypeScript ma, anche in questo caso, il supporto lascia desiderare. Si presuppone però un refactoring generale del codice in TypeScript

Ho scelto di realizzare un bot per evitare di progettare e realizzare elementi come autenticazione ee front-end.

Part I - funzionamento

Evoluzione di un gioco

1. A, B e C avviano il bot con **/start**
    - se A, B o C non possiedono uno username il bot chiede di rieseguire **/start** quando avranno fatto le giuste modifiche.
2. A utilizza il comando **/sfida** per iniziare una nuova partita
    - se A sta già giocando un altro gioco il bot gli dice che prima deve uscire dal gioco attuale
3. Il bot chiede ad A con quante persone vuole giocare
4. A deve rispondere un numero tra 1 e 3
5. Il bot chiede ad A di inviare singolarmente gli username degli altri giocatori, in questo caso B e C
6. Il bot controlla che tutti i giocatori abbiano avviato il bot
    - se così non fosse il bot lo comunica ad A e gli invia il link al bot agli altri utenti.
7. Il bot manda un messaggio a B e C avvisandoli dell'invito a giocare da parte di A
8. Se B e C rispondono entrambi **/enter** il gioco incomincia
    - B e/o C possono scegliere di rispondere **/rifiuta** per non entrare in gioco con A
    - se così fosse A e/o B e/o C vengono avvisati che B e/o C hanno rifiutato
9. il bot avvisa tutti i giocatori che il gioco è cominciato, comunicando chi da li carte e chi inizia.
10. ogni giocatore rivece un messaggio con le carte in tavola, il numero delle proprie scope e il numero di carte nel proprio mazzetto.
11. Ogni giocatore riceve le carte della propria mano come pulsanti al posto della tastiera
    - se il giocatore di turno può bussare viene mostrato anche il pulsante 'bussa'. In caso di bussata il bot controlla che effettivamente il giocatore sia di turno e che possa bussare.
        - se il giocatore ha in mano un 7C il bot gli chiede che valore vuole assegnare al 7
        - l'utente risponde
    - Se così fosse il bot comunica agli altri giocatori lo stato del gioco e la mano dei giocatori che hanno bussato.
12. il giocatore di turno deve giocare una carta mandandola al bot
    - se il giocatore gioca una carta che non ha in mano il bot lo informa dell'errore e lo invita a inviarne un'altra
    - se il giocatore gioca una carta ma non è il suo turno il bot lo informa di cioà e lo inviata ad attendere
13. il giocatore visualizza le possibili prese come pulsanti al posto della tastiera e manda la scelta al bot
    - se il giocatore manda una presa non valida il bot lo informa di ciò e lo invita ad inviarne un'altra
14. il bot informa gli altri giocatori della mossa di chi ha giocato e manda il nuovo stato del gioco
15. Il bot dice ai giocatori chi è di turno
16. Ogni volta che una mano termina il bot informa i giocatori delle numero di mani restanti
17. In punti 10-16 si ripetono finche le carte nel mazzo non finiscono
18. Alla fine del gioco il bot comunica ai giocatori i punti accumulati e annuncia il vincitore

Comandi

comando	descrizione
/start	avvio del bot
/sfida	inizio di una nuova partita
/entra	partecipare ad un gioco su invito
/rifiuta	rifiutare l'invito a un gioco
/esci	uscire dal gioco
/status	visualliza le statische di gioco
/tutorial	come giocare a cirulla
/aiuto	info sui comandi del bot
/privacy	info sulla privacyy dei dati del bot
/info	info sul bot
/stop	arresto del bot

Messaggi dell'utente

Il bot interagisce anche ad altri messaggi da parter dell'utente, non solo comandi.

testo	descrizione
carta	mossa di gioco
bussare	tentativo di bussata

Part II - implementazione

Un bot di Telegram è, alle sue basi, una web API. Ogni volta che viene mandato un messaggio questo viene inoltrato a un server tramite un webhook, viene processato e viene inviata indietro una risposta all'utente. Nel caso di Cirullino mi sono affidato a Telegraf.js per astrarre il concetto di web API e gestire il bot a un livello più alto. Il bot è un'instanza su Heroku sulla quale gira un'applicazione Node.js.

Il framework prevede che il bot reagisca a input da parte dell'utente. Quando il bot riceve un messaggio può rispondere e concludere la conversazione immediatamente oppure intrattenere una conversaione più lunga entrando in un percorso di scene. Nella mia implementazione le scene vengono usate anche per dividere in modo più logico le funzioni del bot. I comandi che richiedono una logica pià elaborati seguono un percorso di scene ben definito.

Dato che il bot deve essere il più indipendente possibile dai dati e dallo stato del gioco ha senso utilizzare un database per persistere i dati. Utilizzando Heroku è molto immediato appoggiarsi a Redis. In Cirullino Redis viene utilizzato solo per ricordare le info sugli utenti e sui giochi attivi. Tutto il resto: dati su giochi passati, statistiche, etc dovranno essere immagazzinati su un'altro DB, per ora si pensa un'istanza di MongoDB Atlas, ma verrà decisio in futuro. Questa scelta ha l'obbiettivo di mantenere le dimensione e la complessità del DB redis al minimo per garantire elevate performance.

Redis structure

id	description	type
users	username : userId pair	HASH
userIds	userId : username	HASH
userInfo	userId : name	HASH
userInvites	points to invite of a user,sorted by userid	ZSET
invites.X	which users accepted the invite X	HASH
invitesIdUsed	free id for the new invite	BITMAP
userActiveGroup	which group a user is playing	HASH
groupIdUsed	know the free id of the group and not waste ids	BITMAP

Game data of groups.1.

id	description	type	reason for type
activeUserId	id of active user	STRING	
mattaValue	value of matta in the game	STRING	
lastWhoTook	userId of last who took cards from board	STRING	
bonusPoints	obj indexed by userId with bonus points	HASH	
isBussing	obj indexed by userId with status of bussata	HASH	
board	cards in the board	SET	no care for order need to always access all
userIds	userIds in group	SET	no care for order need to always access all
hands.userId	cards in the hand of a user	SET	no care for order need to always access all
tricks.userId	possible tricks of userId	SET	no care for order need to always access all
gameDeck	cards in deck	LIST	fast for popping
strongDeck.userId	cards in the strongDeck of a user	LIST	fast for pushing
weakDeck.userId	cards in the weakDeck of a user	LIST	fast for pushing

Basic command behavior

Create a messages.ts to export all useful messages. And a wrapper function that takes as argument the message and sends it back to the user

/help

- reply with help message

/privacy

- reply with privacy message

/about

- reply with about message

/tutorial

- reply with tutorial message

/info

- reply with info message

/status

need to think about stats DB then implement command

/start

- check if user has username
- if so impostare userId of user in db
- else ask him to set it and then resend /start

Advanced commands behavior

/play (A starts a game with B and C)

- if there is an entry in userActiveGroup reply with 'exit current game and then start another one'
- else ask how many players want to play
    - if user response is not a number between 1 and 3 answer notANumberMsg and keep waiting
    - else O is # of opponents
        - O times
            - ask name of opponent X
            - if X not in users hash reply with notStartedBotMsg
            - if X in userActiveGroup reply with userAlreadyPlayingMsg
            - else store username
        - look at the invitesIdUsed, get the index Y of the first 0 and set it to 1
        - for each opponent add a key-value in the userInvites 'userdId : Y'
        - create a new HASH invites.Y with key 'AuserId : true, BuserId : false, CuserId : false'
        - send to opponents inviteReqMsg

/enter inviteId

- if there is not an entry in userInvites with key = userId reply notInvitedMsg
- else if userInvites.userId contains inviteId set value = true in invites.inviteId where key = userId
    - if some value in invites.inviteId is still false reply with stillSomeoneToAcceptMsg
    - else set invitesIdUser to 0 at index = inviteId
        - for each id reamining userInvites at score = userId
            - for each user in invites.id
                - send inviteNoLongerValidMsg
                - remove from userInvites at score = user remove member id
        - for each userId in invites.inviteId remove from userInvites where score = userId the member = inviteId
        - delete invites.inviteId
        - in a transaction
            - get the first available id in groupsUsedIds (Y)
            - create all necessary data structure for a game: groups.Y...
        - send to all users in groups.Y.userIds the initial status of the game

/refuse inviteId

- if there is not a record in userInvites with key = userId (X) reply with info saying that user has not been invited
- else store the users in invites.X
    - for each user
        - delete entry in userInvites
    - send a messagge to all users telling that about rejection

/exit

- check if in userInvites there is a key = userId
    - if there is in a single transaction:
        - look for the corresponding invites.X
        - store the userIds to send them later info about exit
        - for each userId
            - delete the key in userInvites
        - delete invites.X
        - send to other users info about exit
- check if there is an entry in userActiveGroup with the key = userId
    - if there is: make a transaction:
        - look for the corresponding group
        - store userIds to send later info about exit
        - for each userId in the group
            - delete game data of a user
            - delete the entry in the userActiveGroup hash
        - delete game data of the group
        - delete group hash
        - reply to the user he exited
        - send to other users in the game info about exit

User msg behavior

cards - /[A0123456789JQK][♥♦♣♠]/g

- if doesn't exists an entry in userActiveGroup with key = userId reply notPlayingMsg
- else take groupId G
    - if user is not active reply 'userNotActiveMsg'
    - else take user hand
        - if user sent more than 2 cards and the card is not in the hand of the user
            - ckosa
        - else remove card from user hand
            - analyze possible tricks
    - take the hands of the user and if it doesn't include the used card
        - if not check if groups.G.tricks.userId exists
            - if not answer with error info 'not a valid move'
            - if it exists parse it and check if the used card is in one of the tricks
                - if not send back 'not a valid move'
                - else update game status and delete groups.G.tricks.userId
                    - if game is ended update db and tell user outcome of game
                    - else if hand is ended reset isBussing in db, update hands in db
                        - share game status to other users
        - else analyze possible tricks
            - if there are no tricks move is 'calata'
            - if there is only 1 trick directly update game status
            - if there are more than 2 tricks ask user which one to choose

bussata

- check if exists an entry in userActiveGroup
    - if not reply saying the user is not playing
    - else take groupId X and check who is the active user of the game
        - if user is not active send back 'not active user'
        - else check if user is already bussing
        - if yes send back 'already bussing' message
            - else check if user can 'buss'
                - if it can't send back 'can't buss messagge'
                - else check if user has a 7C
                    - if it has, bot asks for the value of the 7C
                    - user sends back value
                    - set matta value in db
                - update bonus points in DB and who is bussing
                - send new game status to other players

middlewares

- [x] updateUserInfo(username,userId)
    - get userId from userIds
        - if does not exists create it and create reverse in users
        - else if old username and current username are different
            - delete old entry from users hash
            - set entry [userId:username]

helper functions

- [ ] getUserActiveGroup(userId)
    - return HGET userActiveGroup userId
- [ ] isUserActive(userId, groupId)
    - return GET groupX.activeUserId == userId
- buildReply (...args, options : IMsgOpions)

bot reply msgs

name	text	args	options
helpMsg	``		
privacyMsg			
aboutMsg			
tutorialMsg			
infoMsg			
notANumberMsg	`Devi dirmi un numero tra 1 e 3`	-	
notStartedBotMsg	`Mi dispiace ma ${user} non ha avviato cirullino, inviagli questo link ...`	user	noLinkPreview

name	text	args	options
userAlreadyPlayingMsg	`Mi dispiace ma ${user} sta già giocando, chiedigli di ternimare il gioco`	user	
notInvitedMsg	`Mi dispiace ma non sei stato invitato da nessuno`		
inviteReqMsg	`Sei stato invitato da ${user}, per entrare rispondimi con '/enter ${inviteId}'`	user inviteId	
stillSomeoneToAcceptMsg	`Perfetto, c'è ancora qualcuno che deve accettare l'invito`		
inviteNoLongerValidMsg	`Mi dispiace ma l'invito ${inviteId} non è più valido perchè ${user} ha cominciato un altro gioco`	user inviteId	
notPlayingMsg	`Mi dispiace ma non stai giocando con nessuno, inizia un gioco con /start`		
userNotActiveMsg	`Mi dispiace ma non è il tuo turno, aspetta`		

Scenes

Typescript interfaces

```
interface IMsgOptions{
  noLinkPreview ?: boolean;
}
```