UNIVERSITÀ
DI TRENTO

Dipartimento di ingegneria e scienza dell'informazione

Progetto:

Vocable

Titolo del documento:

Documento di Architettura

Document info

Doc. Name	D3-Vocable_Architettura	Doc. Number	D3
Description	Il documento include diagrammi delle classi e codice in OCL		

INDICE

Scopo del documento

Il presente documento definisce l'architettura del progetto "Vocable" mediante diagramma delle classi in UML (Unified Modeling Language) e codice in OCL (*Object Constraint Language*). Nel documento precedente (*D2-G48*) sono forniti: diagramma dei casi d'uso, diagramma di contesto e diagramma dei componenti.

Basandosi sui suddetti diagrammi si definisce l'architettura del sistema, di cui si illustrano le classi da implementare a livello di codice e la logica che regolerà il funzionamento del software.

Diagramma delle classi

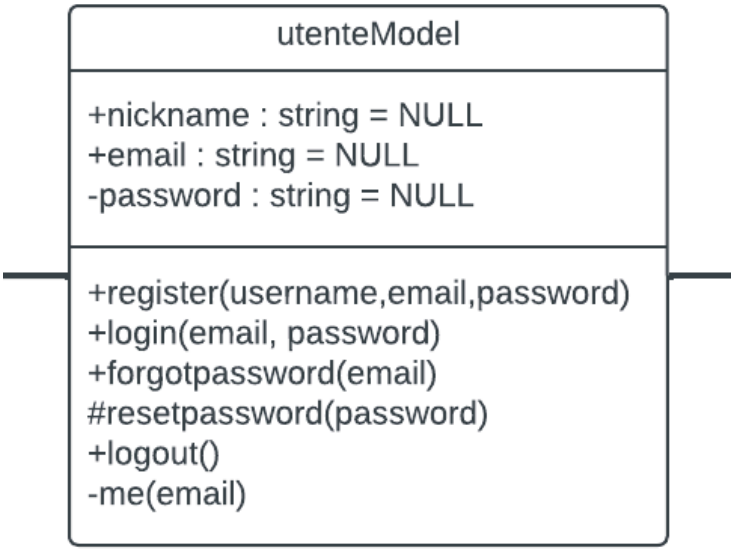
Nel presente capitolo vengono presentati le classi previste nell'ambito del progetto "*Vocable*". Ogni componente presente nel diagramma dei componenti diventa una o più classi. Tutte le classi individuate sono caratterizzate da un nome, una lista di attributi che identificano i dati che vengono gestiti dalla classe stessa e una lista di metodi che definiscono le operazioni interne alla classe. Ogni classe può essere

associata alle altre classi e tramite questa associazione viene definita la relazione tra di esse.

Riportiamo di seguito le classi individuate a partire dai diagrammi di contesto e dei componenti. In questo processo si è proceduto anche nel massimizzare la coesione e minimizzare l'accoppiamento tra classi.

Utenti

Gli utenti dell'applicazione sono identificati dalla classe *utenteModel*, in essa sono contenuti i dati relativi ad ogni account di gioco e le funzioni relative al login e alla modifica della password. La differenziazione tra utenti identificati e anonimi, i quali si distinguono tra loro per diverse interfacce e funzionalità, avviene attraverso il login, la susseguente generazione di un token identificativo conservato nello store e su una gestione dinamica dell'interfaccia grafica basata sulla presenza e validità del token stesso.



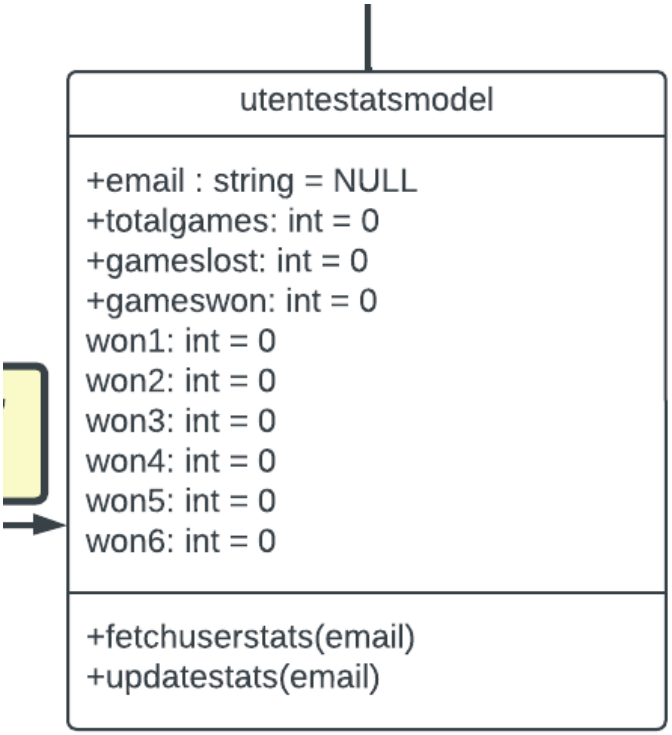
Gestione autenticazione

Il primo sistema subordinato di *Vocable* è quello di gestione credenziali e generazione token. La funzione di login confronta le credenziali immesse dall'utente durante il login con quelle registrate nel database interno contenente le specifiche degli account di gioco Vocable, se tale confronto ha successo, viene generato un token identificativo univoco per l'utente, esso è conservato nello store e autenticato tramite la funzione *authenticateToken*. La corretta autenticazione del token consente

all'utente di rimanere logged-in e di accedere alle funzionalità esclusive dell'interfaccia *id_interface*.

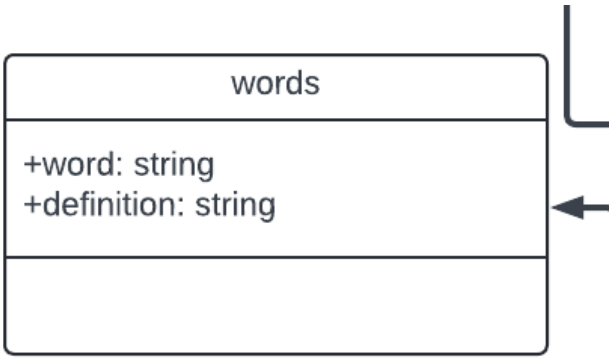
Gestione statistiche

Vocable utilizza un database di gestione statistiche, grazie al quale partite, vittorie e vittorie per numero di tentativi sono calcolate, registrate e rese disponibili per la visualizzazione ai rispettivi utenti.



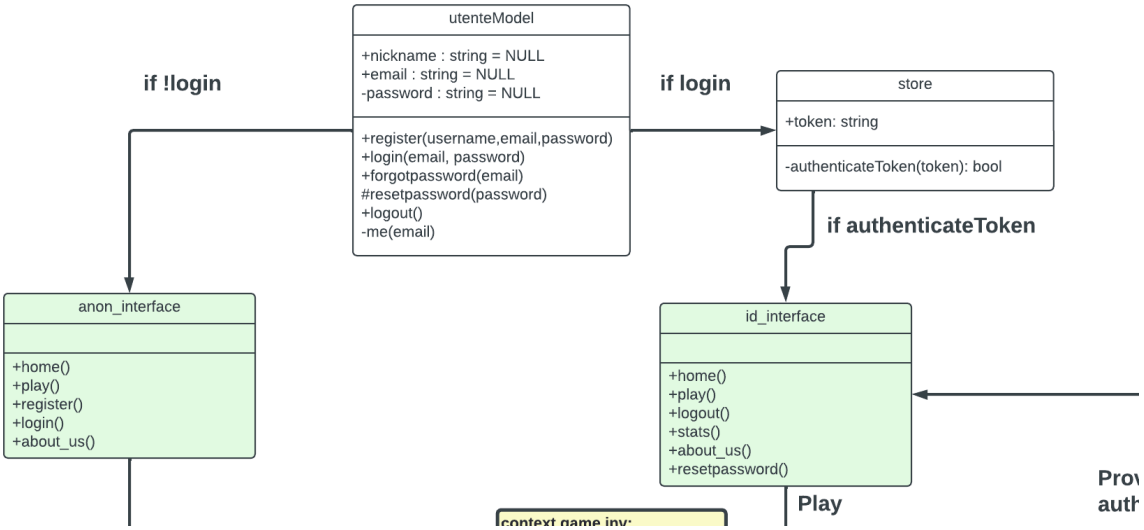
Vocabolario

Al centro della meccanica di gioco di *Vocable* vi è il dizionario. Un file contenente varie parole associate alle rispettive definizioni. La notevole quantità di vocaboli garantisce la non ripetitività del gioco



Schermate Home

Le interfacce per gli utenti anonimi o registrati sono identificate rispettivamente dalle classi astratte *anon_interface* e *id_interface*. Queste forniscono le funzioni necessarie a: iniziare una partita, creare un account, fare login o cambiare password (per gli utenti anonimi) e effettuare il logout o visualizzare le proprie statistiche (per gli utenti identificati).



Gestione partite

Infine è presentata la classe contenente i metodi di gestione partita. Essa fornisce agli utenti l'interfaccia di gioco e tutte le funzioni necessarie a completare una partita e a registrarne le statistiche se si dispone di un account *Vocable*. Tali funzioni comprendono: l'effettuazione di tentativi per indovinare la parola nascosta, la fornitura di indizi ai giocatori a ogni tentativo incorretto, l'output di notifiche di vittoria

o sconfitta, il salvataggio automatico delle statistiche sull'account dell'utente registrato e la possibilità di iniziare una nuova partita.

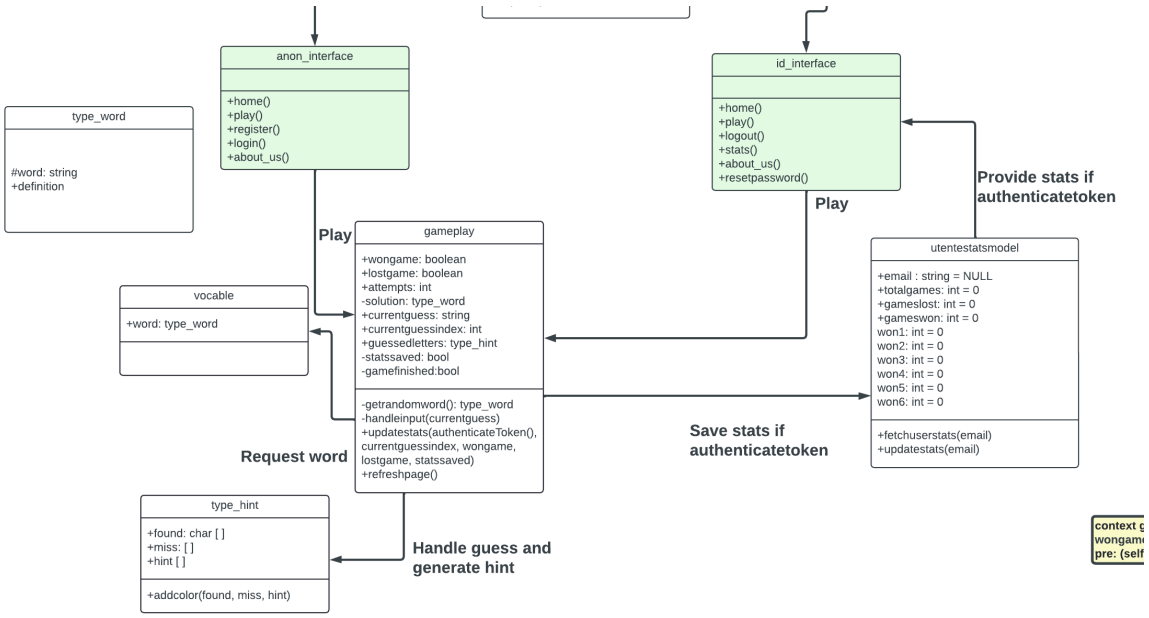
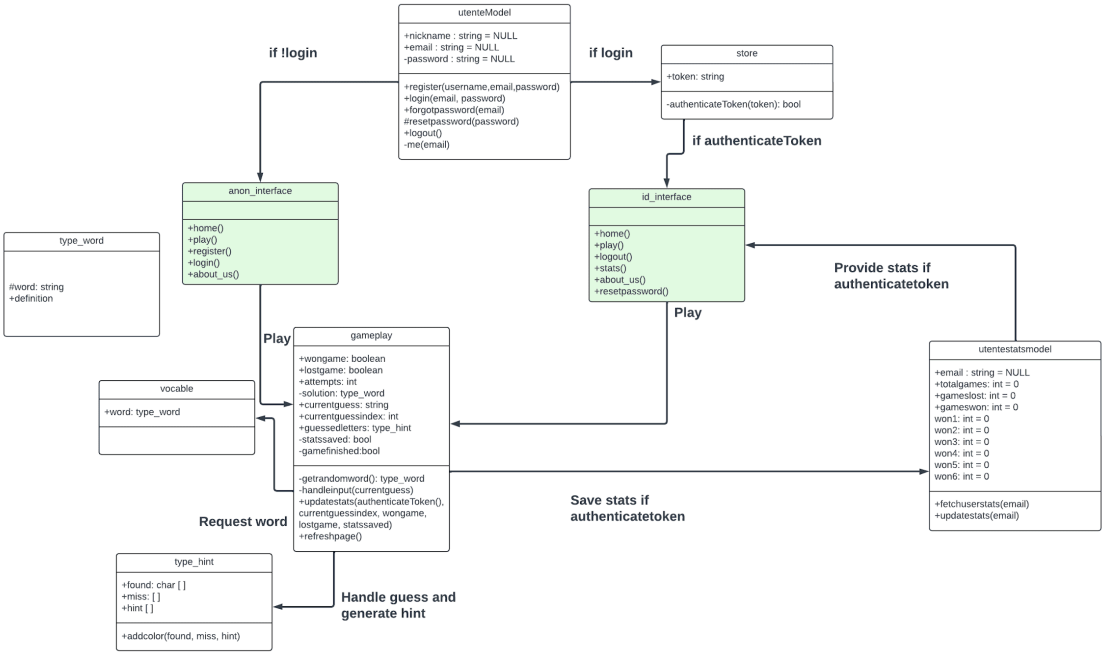


Diagramma delle classi complessivo

In seguito è riportato il diagramma completo, costituito da tutte le classi precedentemente presentate.

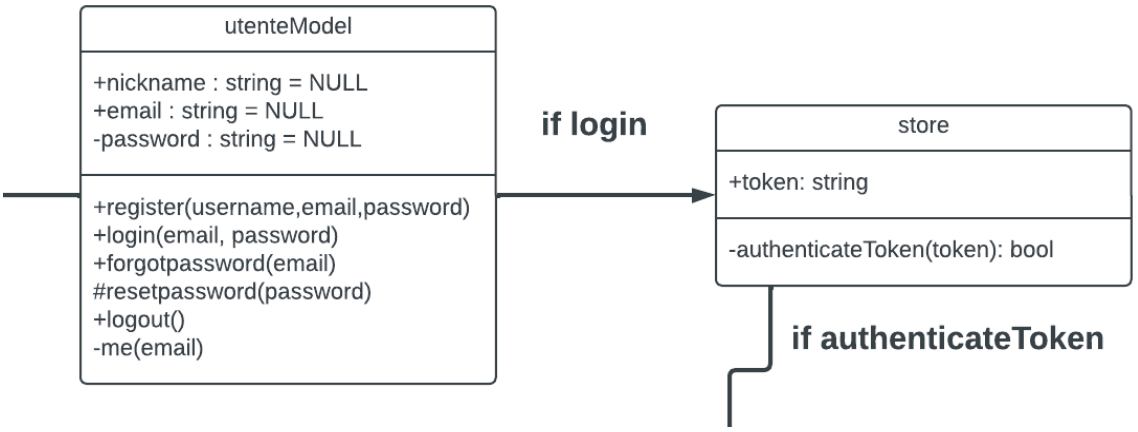


Codice in Object Constraint Language

In questa sezione del documento viene descritta in modo formale la logica prevista nell'ambito di alcune operazioni di alcune classi. Questa logica viene descritta tramite il linguaggio "*Object Constraint Language*" (*OCL*), poiché tali concetti non sono esprimibili in nessun'altra forma nel contesto di *UML*.

Login

Nella classe utentemodel è presente il metodo *login()*. L'esecuzione di questo metodo comporta, oltre al salvataggio delle informazioni dell'utente, la generazione ed identificazione di un token.
Questa condizione, presente nelle classi raffigurate qui di seguito:

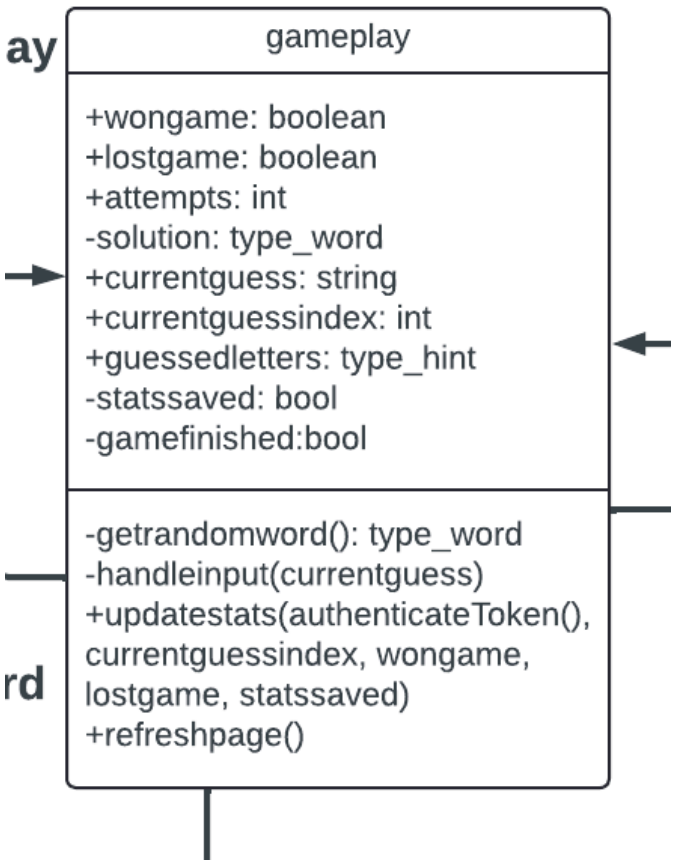


è espressa in OCL tramite questo codice:

```
context utenteModel inv: let token :
string = self.login() in
store.authenticateToken(token) = true)
```

Condizioni per i Games

Nella classe games ci sono due condizioni che devono essere verificate per ogni istanza della classe. Queste sono: il numero di tentativi per indovinare la parola deve essere al massimo 6 e la lunghezza della parola da indovinare deve essere pari a quella dei tentativi immessi dall'utente.
La classe a cui fanno riferimento queste condizioni è la seguente:



e sono dunque esplicitate in codice OCL come il codice qui inserito:

```
context game inv:
currentguessindex <= 6
strlen(currentguess) ==
strlen(solution.word)
```

Completion in game

Nella classe appena vista (*2.2)*, sono presenti delle costrizioni dell'attributo "*gamefinished*".
Tali costrizioni sono le seguenti: *gamefinished* può assumere il valore '1' se e solo se l'utente indovina la parola o i tentativi per indovinare la parola in questione finiscono.
Queste costrizioni possono essere espresse in *OCL* in questo modo:

```
context game inv:
if (self.currentguessindex >=
7 or wongame) then
gamefinished = true
```

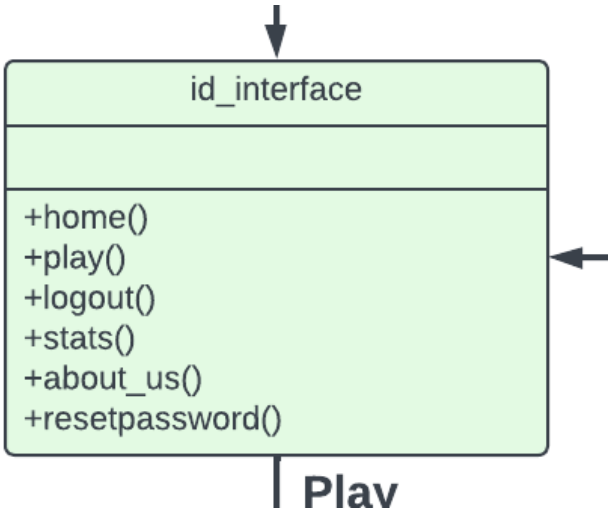
Salvataggio statistiche in games

Nella stessa classe (*2.2*) è presente anche la funzione "*updatestats*". Tale funzione può essere eseguita solo se la variabile "*gamefinished*", presente nella stessa classe, corrisponde al valore '1' e, in contemporanea, anche quando la funzione "*authenticateToken*" restituisce 'true'
Questa condizione è espressa tramite una precostrizione come la seguente:

```
context game::updatestats(authenticateToken(), currentguessindex,
wongame, lostgame, statssaved)
pre: (self.gamefinished == true) AND (store.authenticateToken == true)
```

Logout

Nella classe "i*d_interface*" è presente il metodo *"logout()*".
Questo metodo, dopo essere stato eseguito, prevede la cancellazione del token identificativo e, di conseguenza, la restituzione dalla funzione *authenticateToken* di un valore "false".
Tale condizione, presente nella seguente classe:

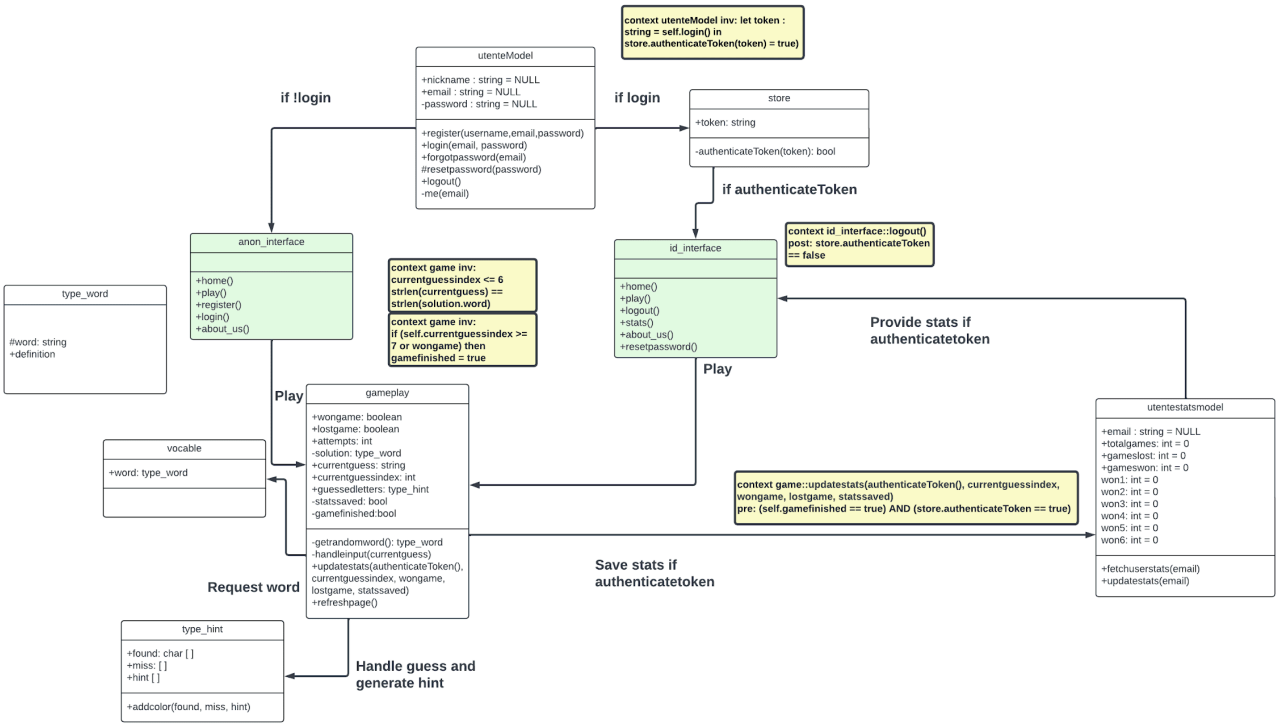


è codificata in OCL tramite una postcondizione, scritta in questo modo:

```
context id_interface::logout()
post: store.authenticateToken
== false
```

Diagramma delle classi con codice OCL

Riportiamo, infine, il diagramma delle classi col codice OCL visto fino ad ora:


context utenteModel inv: let token : string = self.login() in store.authenticateToken(token) = true)
utenteModel
+nickname : string = NULL
+email : string = NULL
-password : string = NULL
+register(username,email,password)
+login(email, password)
+forgotpassword(email)
#resetpassword(password)
+logout()
-me(email)
if !login
if login
store
+token: string
-authenticateToken(token): bool
if authenticateToken
anon_interface
+home()
+play()
+register()
+login()
+about_us()
context id_interface::logout()
post: store.authenticateToken == false
id_interface
+home()
+play()
+logout()
+stats()
+about_us()
+resetpassword()
type_word
#word: string
+definition
context game inv:
currentguessindex <= 6
strlen(currentguess) == strlen(solution.word)
context game inv:
if (self.currentguessindex >= 7 or wongame) then gamefinished = true
Provide stats if authenticatetoken
Play
Play
gameplay
+wongame: boolean
+lostgame: boolean
+attempts: int
-solution: type_word
+currentguess: string
+currentguessindex: int
+guessedletters: type_hint
-statssaved: bool
-gamefinished:bool
-getrandomword(): type_word
-handleinput(currentguess)
+updatestats(authenticateToken(), currentguessindex, wongame, lostgame, statssaved)
+refreshpage()
vocable
+word: type_word
utentestatsmodel
+email : string = NULL
+totalgames: int = 0
+gameslost: int = 0
+gameswon: int = 0
won1: int = 0
won2: int = 0
won3: int = 0
won4: int = 0
won5: int = 0
won6: int = 0
+fetchuserstats(email)
+updatestats(email)
context game::updatestats(authenticateToken(), currentguessindex, wongame, lostgame, statssaved)
pre: (self.gamefinished == true) AND (store.authenticateToken == true)
Save stats if authenticatetoken
Request word
type_hint
+found: char [ ]
+miss: [ ]
+hint [ ]
+addcolor(found, miss, hint)
Handle guess and generate hint