Progetto d'esame di Tecnologie e Applicazioni Web (2017-2018)

Veneri Alberto

Architetture del sistema

Introduzione

Come da specifiche è stata realizzata una web application utilizzando il cosiddetto stack MEAN (MongoDB, Express, Angular e Node.js). Il sistema è stato scomposto in due parti fondamentali: l'applicativo lato server e le applicazioni lato client. L'applicativo lato server, realizzato in Node.js, fornisce delle API in stile REST (REpresentational State Transfer) per interagire con i dati presenti all'interno di un database realizzato con MongoDB da un qualsiasi client in grado di eseguire delle richieste HTTP. Le applicazioni lato client create realizzate sono state 3: una web app, una applicazione Android e una applicazione Desktop. Le applicazioni per i tre tipi di piattaforma sono identiche dal punto di vista della UI (User Interface) e sono tutte basate sul codice scritto per la realizzazione della SPA (Single Page Application) in Angular. Uno deployment diagram riassuntivo dell'architettura dell'applicazione è fornito nella Figura 1.

Applicazione lato server

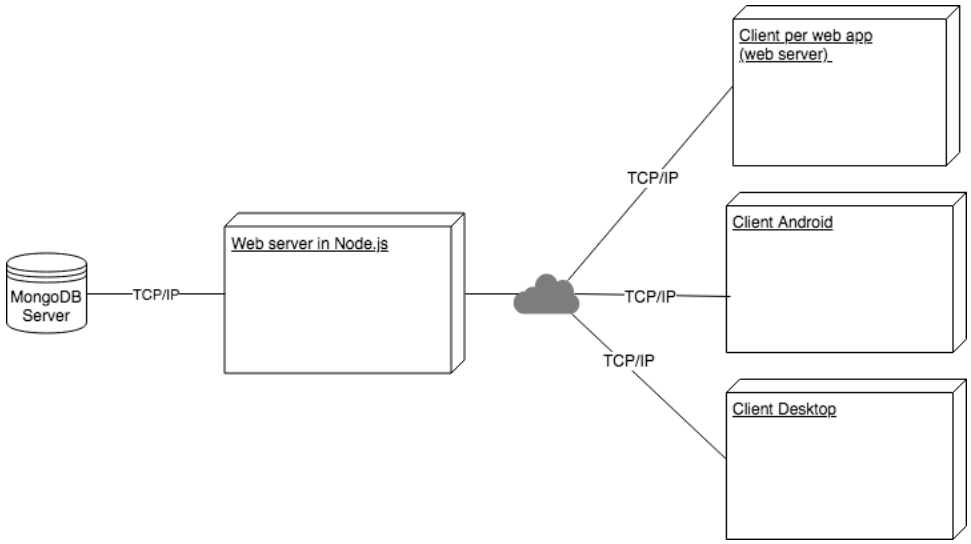
L'applicazione lato server è stata realizzata attraverso la piattaforma Node.js ed è composta da 6 classi e 1 file "main" che quando eseguite mettono a disposizione sulla porta 8080 della macchina in cui sono stati avviati una serie di API REST che permettono di interagire con il database in MongoDB collegato. Tutto il codice dell'applicazione è stato scritto in TypeScript per cercare di struttura il codice meglio ed aumentarne la leggibilità rispetto ad un'applicazione scritta in JavaScript "puro". Ricordiamo che TypeScript è un linguaggio che estende JavaScript ed è transpilato[1] in codice JavaScript quindi viene eseguito da un qualsiasi motore JavaScript; in particolare Node.js utilizza il motore JavaScript V8, sviluppato da Google e rilasciato sotto licenza BSD.
Il routing per le API è stato realizzato attraverso il framework express.

Applicazioni lato client

Le applicazioni lato client sono basate sulla piattaforma open source Angular e forniscono all'utente finale una semplice interfaccia per interagire con le API fornite dall'applicazione lato server. Angular è una piattaforma per lo sviluppo di applicazioni web di tipo SPA ed è facilmente integrabile con i framework Apache Cordova ed Electron per la realizzazione di applicazioni per smartphone e per desktop.

[1] Compilazione da un sorgente ad un altro sorgente

Figura 1: Deployment diagram



Modello dei dati utilizzato

L'applicazione si appoggia ad un DBMS non relazionale: MongoDB. MongoDB è un DBMS (DataBase Management System) orientato ai documenti e rappresenta una delle tecnologie più interessanti fra i NoSQL DBMS e sta diventando ormai uno standard de facto nelle web application moderne. In MongoDB i documenti sono rappresentati come oggetti JSON il che rende questo DBMS ben integrabile con JavasScript. All'interno della nostra applicazione abbiamo utilizzato MongoDB attraverso l'ODM (Object Dcoument Mapper) Moongose che ha reso possibile il recupero, l'inserimento e l'aggiornamento dei dati all'interno di MongoDB con una mappatura automatica all'interno di oggetti JavaScript.

Le collezioni

Per la realizzazione della web application sono state realizzate 5 collezioni: messages, chats, fields, matches e users. Nel seguito sono descritte le strutture delle collezioni, per semplicità sono stati omessi i campi "_id" e "__v" che rappresentano rispettivamente i campi chiave che sono indicizzati e creati automaticamente da MongoDB e il campo che rappresenta la versione delle API.

Messages

Messages rappresenta la collezione che contiene i dati dei messaggi appartenenti alle chat degli utenti.

Campo	Tipo	Descrizione
idChat	ObjectID	Id della *Chat* a cui appartiene il messaggio.
sentAt	ISODate	Data e ora di invio
text	String	Testo del messaggio
senderID	ObjectID	ID dello *User* che ha inviato il messaggio.

Chats

Chats rappresenta la collezione che contiene le informazioni delle chat dei vari utenti.

Campo	Tipo	Descrizione
listMessage	Array di ObjectID	Contiene il una array di *Message* che appartengono alla Chat. Non sarebbe necessario in quanto si potrebbe recuperare dai *Message*, però semplifica la query di recupero dei *Message* a partire da una *Chat*
user1ID	ObjectID	ID di un *User* della *Chat*
user2ID	ObjectID	ID dell'altro *User* della *Chat*
createdAt	IsoDate	Data di creazione della chat

Fields

Fields rappresenta i campi delle partite, ogni campo è associato ad un giocatore e viene creato nel momento in cui un giocatore entra in una partita.

Campo	Tipo	Descrizione
playerId	ObjectID	ID dello *User* a cui è associato il campo

matrix	Matrice 10x10 di oggetti Json così formati:
	{
	"hit": < true/false>,
	"color": < valore esadecimale>
	}
		Matrice che rappresenta il campo. Per ogni cella si ha l'informazione riguardante lo stato colpito/non colpito e un utile codice per capire lo stato della cella: acqua, presenza della

		nave, colpita, colpita e affondata

shipst	Array di oggetti così composti:
	{
	"cells": [
	{
	"x": <x>,
	"y": <y>,
	"hit": <true/false>
	},…
	]
	}
		Array contenente i dati riguardanti le navi. Per ogni nave è presente un array di di oggetti nei quali la "x" corrisponde alla riga e la "y" alla colonna della matrice in cui la nave è presente. Inoltre, "hit" rappresenta se quella cella è stata o meno colpita.

aliveShips	Integer	Numero di navi non ancora affondate

Matches

Matches è la collezione che rappresenta i match fra i giocatori

Campo	Tipo	Descrizione
timestamp	IsoDate	Data e ora inizio del gioco
owner	ObjectId	ID dello *User* che ha creato la partita
opponent	ObjectId	ID dello *User* avversario, riempito solamente quando *status* >= 1
status	Integer	Stato della partita: • 0: in attesa del secondo giocatore • 1: In attesa dello schieramento delle navi dei giocatori • 2: in gioco • 3: partita terminata
lastIdAttacker	ObjectId	ID dell'ultimo *User* che ha attaccato. Durante l'inizializzazione del gioco viene settato al valore dell'Id dell'avversario dello *User* che dovrà iniziare la partita
fieldOpponent	ObjectId	Riferimento al *Field* del giocatore *opponent.* Riempito solamente quando *status* >= 2

fieldOwner	ObjectId	Riferimento al *Field* del giocatore *owner*. Riempito solamente quando *status* >= 2

User

Questa collezione rappresenta gli utenti che si sono registrati all'applicazione

Campo	Tipo	Descrizione
mail	string	Mail dell'utente
name	string	Nome dell'utente
surname	string	Cognome dell'utente
username	string	Username dell'utente
salt	string	Salt per il calcolo dell'HMAC della password
digest	string	Digest dell'HMAC calcolato
partitePerse	Integer	Numero partite perse dall'utente
partiteVinte	Integer	Numero partite vinte dall'utente
isAdmin	Boolean	Campo che rappresenta l'appartenenza o meno dell'utente al gruppo di utenti amministratori
chatList	Array di ObjectId	Array di ID delle *Chat* a cui a ha partecipato/partecipa l'utente

REST API

Di seguito è riportata una tabella contenente la descrizione dei vari endpoint disponbili dal server web realizzato con Node.js.

ENDPOINT	METODO	JSON da inviare	Autenticazione	Struttura della risposta	Descrizione
/	GET	Nessuno	Nessuna	{ "api_version": "1.0", "endpoints": ["/users", "/chats", "/scoreboard", "/matches"] }	Ritorna la lista delle root degli endpoint principali.
/users	POST	{ "username":<username>, "name":<nome> "surname":<cognome>, "password":<password>, "mail":<email> }	Nessuna	{"error": <true/false>, "errormessage":<error message o “”>, "id":<id nuovo utente> }	Crea un nuovo utente e ritrona un Json contente l'id del nuovo utente
/login[?remindMe=true]	GET	Nessuno	Basic auth con nome utente e password	{ "error": <true/false>, "errormessage": <error message o “”>, "token": <JWT> }	Login da parte dell'utente tramite basic auth (ipotizzando una connessione in HTTPS) e ritorna un JWT per l'utente
/renew	GET	Nessuno	JWT	{ "error": <true/false>, "errormessage": "", "token": <nuovo token>}	Rinnova il JWT dell'utente già loggato solo nel caso in cui il parametro remindMe=true sia

					stato passato nella query string durante il login

/users/:username	GET	Nessuno	JWT	{
				"chatList":[<chat>,…],
				"_id":<id utente>,
				"name":<nome utente>,
				"surname":<cognome>,
				"username":<username>,
				"mail":<email>,
				"partiteVinte":<numero partite vinte>,
				"partitePerse":<numero partite perse,
				"isAdmin":<true/false>
				}
					Restituisce la maggior parte delle informazioni dell'utente desiderato in base allo username passato

/users/[?keysearched=nome_utente]	GET	Nessuno	JWT	[{ "mail": <email>, "name": <nome>, "surname": <cognome>, "username": <username>, "partitePerse": <numero partite perse>, "partiteVinte": <numero partite vinte>, "isAdmin": <true/false> }, …]	Restituisce la lista di tutti gli utenti e in caso il parametro keysearched sia settato, filtra quelli che hanno un nome simile a quello indicato
/users/:username/matches	GET	Nessuno	JWT	[{ <tutte le informazioni di un match, vedere entpoint GET /matches/:id> },..]	Ritorna la lista dei match dell'utente

/users/:username	PUT	{
		"mail": <email>,
		"name": <nome>,
		"surname":<cognome>,
		"username":<username>,
		"partitePerse":<numero partite perse>,
		
		"partiteVinte":<numero partite
			JWT	{
				"error": <true/false>,
				"errormessage": <messaggio di errore o "">>
				}
					Aggiorna le info dell'utente
					Solo se il JWT appartiene ad un admin si può modificare anche il ruolo. Ogni utente normale può modificare

		vinte>, "isAdmin":<true/false> }			solamente il suo utente, ogni utente admin può modificare tutti gli altri utenti

/users/:username	DELETE	Nessuno	JWT	{
				"error": <true/false>,
				"errormessage": <messaggio di errore o "">
				}
					Eliminare utente. Può essere fatto da un admin oppure dall'utente stesso

/chats/	GET	Nessuno	JWT	[
				{
				"chatList": [
				{
				"listMessage": [<id messaggio>,..],
				"_id": <id della chat>,
				"user1ID": {
				"_id": <id utente 1>,
				"username": <username utente 1>
				},
				"user2ID": {
				"_id": <id utente 2>,
				"username": <username utente 2>
				},
				"createdAt": <Iso Date creazione>,
				"__v": <versione della chat, da rimuovere>
				},…]
				}
				]
					Restituisce tutte le chat dell'utente autenticato

/chats/:id	GET	Nessuno	JWT	[{ listMessage": [{ "idChat": <id Chat>, "sentAt": <ISO Date Invio> "text": <testo del messaggio>, "senderID": <id dell'utente mittente>	Restituisce la chat avente l'id desiderato. È possibile visualizzare solamente le chat a

				},..],
				"_id": <id della chat>,
				"user1ID": {
				"_id": <id utente 1>,
				},
				"user2ID": {
				"_id": <id utente 2>
				},
				"createdAt": <Iso Date creazione>,
				"__v": <versione della chat, da rimuovere>
				}
				]
					cui si ha partecipato

/chats/	POST	{
		"destinatario": <username destinatario>
		}
			JWT	{
				"error": <true/false>,
				"errormessage": <messaggio di errore o "">,
				"id" <id nuova chat>
				}
					Crea la chat con l'utente desiderato. Se si cerca di creare una chat con un utente per il quale esiste già un chat viene restituito un messaggio d'errore

/chats/:id	POST	{
		'sentAt': date,
		'text': text
		}
			JWT	{
				"error": <true/false>,
				"errormessage": <messaggio di errore o "">
				}
					Aggiunta di un messaggio alla chat con id uguale a quello passato come *:id* nell'endpoint

/chats/:id	DELETE		JWT	{
				"error": <true/false>,
				"errormessage": <messaggio di errore o "">
				}
					Cancella la chat con id uguale a quello passato come *:id* nell'endpoint

/scoreboard[?limit=n&type=partiteVinte|partitePerse|total]	GET	Nessuno	JWT	[{ "_id": <id utente>, "username": <username>,	Ritorna della classifica degli utenti in base al

				<campo richiesto>: <valore del campo> },…]	numero di vittorie (campo richiesto sarà uguale a numero vittore). Se si vuole specificare un tipo di classifica diversa bisogna specificarlo nei parametri, secondo la struttura presentata nella colonna endpoint. Con total, si intende il numero di partite effettuate dall'utente.

/matches	GET	Nessuno	JWT	[
				{
				"_id": <id match>,
				"timestamp": <ISO Date di creazione della partita>,
				"owner": {
				"_id": <user id>,
				"username": <user username>
				},
				"status": <stato della partita>,
				"__v": <versione delle API, da rimuovere>
				}
				]
					Restituisce tutti i match in attesa

/matches	POST	Nessuno	JWT	{
				"error": <true/false>,
				"errormessage": <messaggio di errore o "">,
				"id" <id nuovo match>
				}
					Crea un match e lo mette in attesa, il proprietario sarà l'utente che lo ha

					creato. Un giocatore non può avere più di un match in stato di attesa
/matches/:id_match/join	PUT	Nessuno	JWT	`{ "error": <true/false>, "errormessage": <messaggio di errore> | "" }`	Aggiunge il secondo utente alla partita specificata nell'endpoint (*:id*)
/matches/:id_match/board	PUT	`{ "positioning": { "ships": [[{"x":<x>,"y":<y> },..],...] } }`	JWT	`{ "error": <true/false>, "errormessage": <messaggio di errore> | "" }`	Disposizione delle navi, la prima volta che viene chiamato si setta in "building" e una volta che entrambi i giocatori han riempito il campo passa in "active". Se il match è in "attivo" ritorna un errore
/matches/:id_match[?type=fullInfo]	GET	Nessuno	JWT	`{ "_id": <id match>, "timestamp": <ISO Date della creazione del match>, "owner":<ID user che ha creato la partita>, "status": <Intero che rappresenta lo stato della partita>, "__v": <versione delle API, da rimuovere>, "lastIdAttacker": <Id ultimo user che ha attaccato>, "opponent":<ID user avversario>, "fieldOwner":<ID campo dell'owner>,`	Restituisce tutte le info del match. Con il parametro type=fullInfo nella query vengono popolati i campi collegati agli Object ID del campo e degli utenti.

				"fieldOpponent":<ID campo dell'opponent> }	

/matches/:id_match	PUT	{
		"position": {
		"x": <x>,
		"y": <y>
		}
		}
			JWT	{
				"error": <true/false>,
				"errormessage": <messaggio di errore> | ""
				}
					Endpoint per attaccare l'avversario in posizione x, y.

Autenticazione degli utenti

Registrazione

Il processo di autenticazione degli utenti inizia con la registrazione dell'utente. All'avvio dell'applicazione, l'utente viene ridiretto alla pagina di autenticazione e se non è in possesso delle credenziali di accesso deve eseguire la registrazione per poter accedere alla web app. Cliccando sul pulsante "Registrati", viene l'utente viene rediretto alla pagina /signup dell'applicazione, dove deve inserire obbligatoriamente i campi:

- Nome
- Cognome
- Username
- Indirizzo Email
- Password
- Ripeti password

Come mostrato in Figura 2

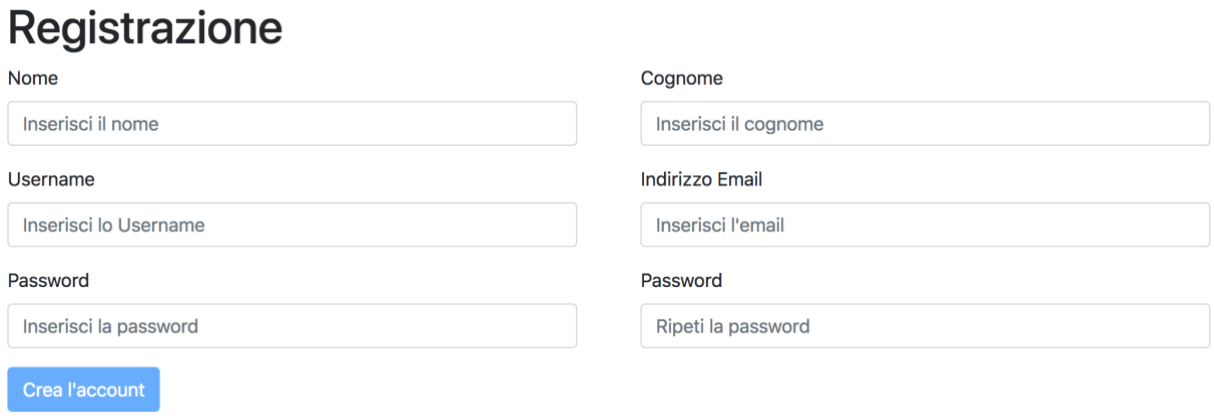


Figura 2: Form di registrazione

Cliccando sul pulsante "Crea l'account" viene inviato un JSON all'endpoint */users* del server[2] con la seguente struttura:

```
{
        "username":<username>,
        "name":<nome>
        "surname":<cognome>,
        "password":<password>,
        "mail":<email>
}
```

Il server verifica che non ci siano errori e/o problematiche nella creazione di un nuovo utente con quelle credenziali e invia una risposta così strutturata:

```
{
        "error": <true/false>,
        "errormessage":<error message> | "",
        "id":<id nuovo utente>
}
```

I campi della risposta hanno la seguente sematica:

- error: ha come valore true se il server ha rilevato una problematica nella creazione dell'utente, false altrimenti

[2] Con il termine server d'ora in poi intenderemo il server realizzato in Node.js che offre le API REST

- errormessage: se error ha valore true, rappresenta il messaggio di erorre, altrimenti non è un campo significativo
- id: in caso non ci siano stati errori, viene ritornato la rappresentazione in stringa dell'ObjectId del nuovo utente creato

In Figura 3 è rappresentato tramite un sequence diagram l'interazione fra i vari utenti.

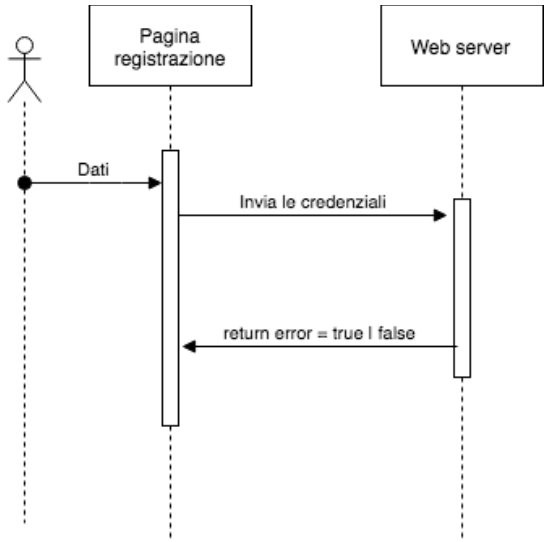


Figura 3: Sequence diagram della registrazione di un utente

Login

Dopo aver effettuato la registrazione un utente può effettuare l'accesso alla web application attraverso la pagina /login (Figura 4)

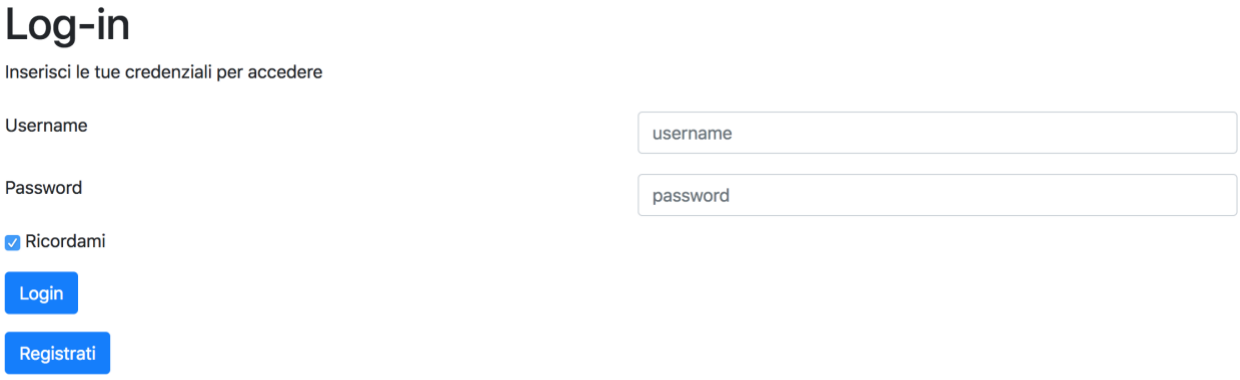


Figura 4: Pagina di login

Il Workflow del login segue i seguenti step:

1. inserimento di nome utente e password e click su "Login" da parte dell'utente
2. il client invia una richiesta all'endpoint /login del server con Metodo GET e inserendo le credenziali nell'header per richiedere una Basic Authentication.
   La Basic Authentication avviene semplicemente inserendo nell'header della richiesta

HTTP la chiave *Authorization* con il valore *<credenital>* dove *<credential>* è semplicemente la codifica in Base64 di *username:password.*
In caso per esempio si fosse registrato un utente con username: mario e password: vfH5y4&Lkft7, l'header risultante sarebbe il seguente:
Authorization: Basic bWFyaW86dmZINXk0JkxrZnQ3

3. il server riceva la richiesta e verifica se le credenziali inviate sono effettivamente presenti nel database. Nel nostro caso abbiamo utilizzato il middleware per express http-passport che ci ha semplificato l'acquisizione delle informazioni dall'header HTTP. Alla fine viene ritornata la risposta da parte del server con un json con la seguente struttura:
```
{
"error": <true/false>,
"errormessage": <error message> | "",
"token": <JWT>
}
```
Per le chiavi *error* ed *errormessage* valgono le stesse considerazioni fatte in precedenza per la registrazione. Mentre il campo token rappresenta il JWT (JSON Web Token) ritornato in caso di autenticazione riuscita.

Come ripetuto più volte durante il corso, siamo a conoscenza che la tecnica è altamente rischiosa se fatta attraverso una richiesta HTTP standard e non HTTPS perché espone il nome utente e la password in chiaro a possibili attacchi del tipo "man in the middle". Per semplificazione comunque abbiamo deciso di passarle in una richiesta HTTP standard essendo questo un progetto a fini didattici e che non richiede la pubblicazione online.
Alla fine del processo di autenticazione il client ha ricevuto un JWT che servirà come lasciapassare per l'accesso alla pagina agli utenti registrati. Nel JWT ritornato sono contenute le seguenti informazioni:

- nell'header sono contenute le informazioni per la verifica della firma del JWT:
```
{
   "alg": "HS256",
   "typ": "JWT"
}
```
- nel payload sono inserite le informazioni riguardante l'utente
```
{
   "username": <username dell'utente>,
   "isAdmin": <valore booleano>,
   "mail": <email>,
   "id": <id dell'utente>
   "name": <nome dell'utente>,
   "surname": <cognome dell'utente>,
   "remindMe": <valore booleano per il rinnovo o meno del
               token>,
   "iat": <timestamp emissione token>,
   "exp": <timestamp scadenza token>
}
```
- E infine la firma digitale del token, del tipo:
HMACSHA256(base64(header) + "."+ base64(payload), secretkey)

Il JWT per essere riutilizzato viene salvato nel locale storage del client.

Descrizione del client web

Per lo sviluppo del client web è stato utilizzato il framework Angular e per realizzazione di tutte le features richieste sono stati creati 12 component (oltre a quello principale per l'avvio dell'applicazione) e 4 servizi.

Services

I servizi in Angular si occupano dell'accesso, il recupero e il salvataggio di informazioni (solitamente da fonti esterne). Ecco una breve descrizione dei servizi creati per lo sviluppo dell'applicazione.

Utilities

Servizio generico che offre principalmente due funzioni: ritornare la base url dell'applicazione lato server e creare l'header HTTP utilizzato dagli utenti loggati.

User

Si occupa di tutte le interazioni con il server per la gestione dell'utente: login, logout, registrazione, rinnovo del token, aggiornamento delle informazioni degli utenti, eliminazione di un utente, recupero delle informazioni di un utente, recupero delle chat di un utente, …

Socket

Si occupa della gestione di un socket creato con il package socket.io

Match

Servizio per la gestione del match, viene utilizzato durante la partita per ricevere informazioni sul match e per aggiornare il campo di conseguenza

Components

I componenti in Angular si occupano della presentazione dei dati e non del loro recupero. Essendo facilmente leggibili e non complicati per la loro realizzazione dal punto di vista logico, ne riepilogo solamente la lista:

- AppComponent
- UserLoginComponent
- UserInfoComponent
- UserSignupComponent
- NavbarComponent
- ScoreboardComponent
- PlayersComponent
- ListChatsComponent
- MatchBuilderComponent
- ChatComponent
- NotfoundComponent
- ListMatchesComponent
- MatchComponent

Routes

Per semplicità riporto la lista delle routes attraverso la costante “routes” che viene utilizzata per il routing ed è, a mio avviso, autoesplicativa:

```
const routes: Routes = [
  { path: '', redirectTo: '/login', pathMatch: 'full' },
  { path: 'login', component: UserLoginComponent },
  { path: 'user', component: UserInfoComponent },
  { path: 'user/:username', component: UserInfoComponent },
  { path: 'user/:username/matches', component:
ListMatchesComponent },
  { path: 'signup', component: UserSignupComponent },
  { path: 'scoreboard', component: ScoreboardComponent },
  { path: 'players', component: PlayersComponent },
  { path: 'chats', component: ListChatsComponent },
  { path: 'chats/:id', component: ChatComponent },
  { path: 'match', component: ListMatchesComponent },
  { path: 'match/:id/board', component: MatchBuilderComponent},
  { path: 'match/:id', component: MatchComponent },
  { path: '**', component: NotfoundComponent }
];
```

Workflow tipico dell'applicazione

In questa sezione riassumerò il workflow tipico dell'utilizzo dell'applicazione attraverso la simulazione di una partita fra due giocatori.

Login

Figura 5: Login client 1

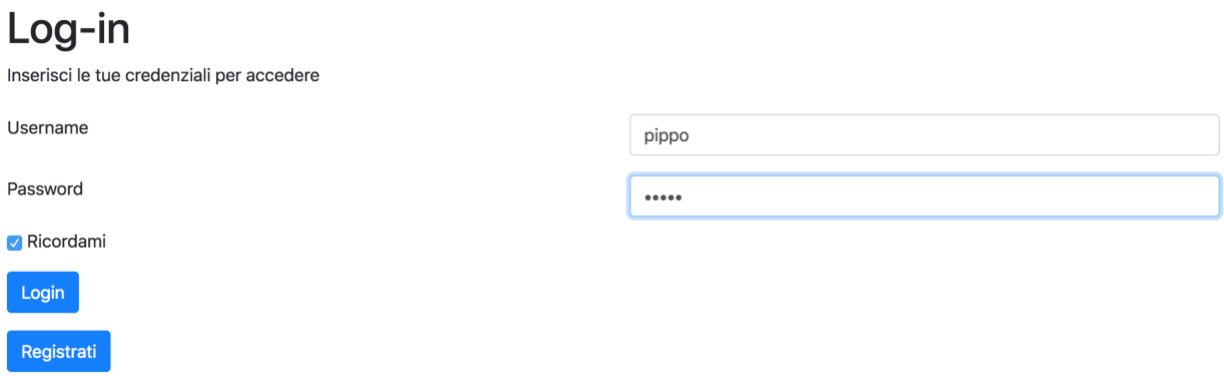


Figura 6: Login client 2

Creazione partita

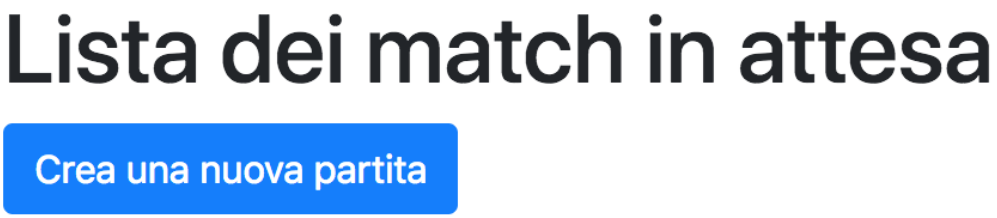


Figura 7: Creazione di una nuova partita, client 1

Figura 8: Client 1 in attesa del secondo giocatore

Posizionamento navi

Figura 9: Join alla partita del client 2

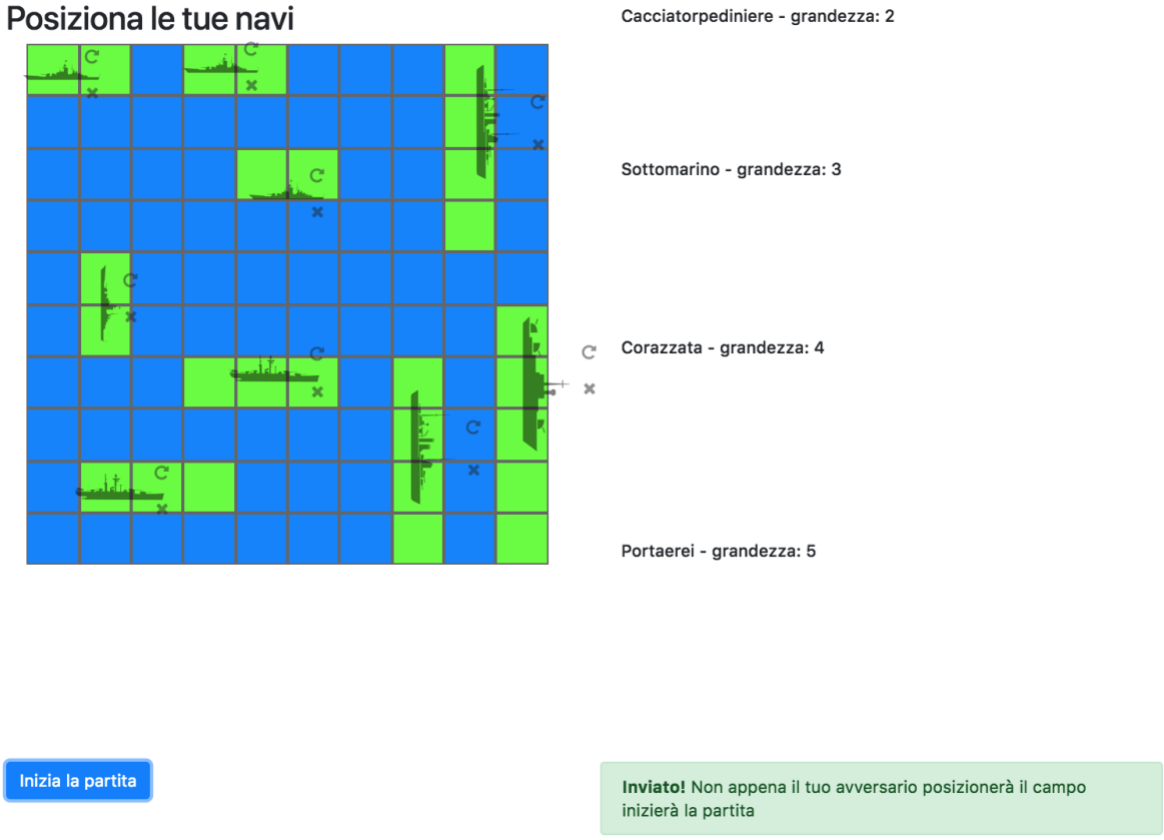


Figura 10: Disposizione delle navi da parte del client 2

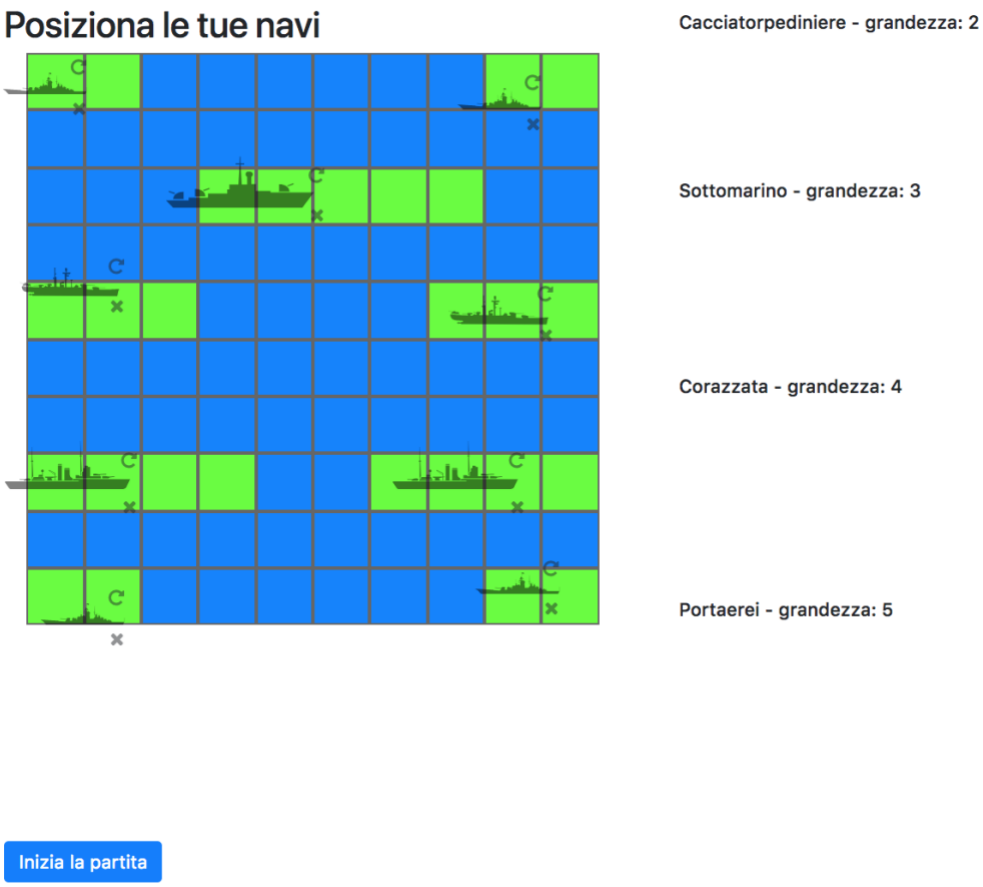


Figura 11: Disposizione navi da parte del client 1

Fase principale di gioco

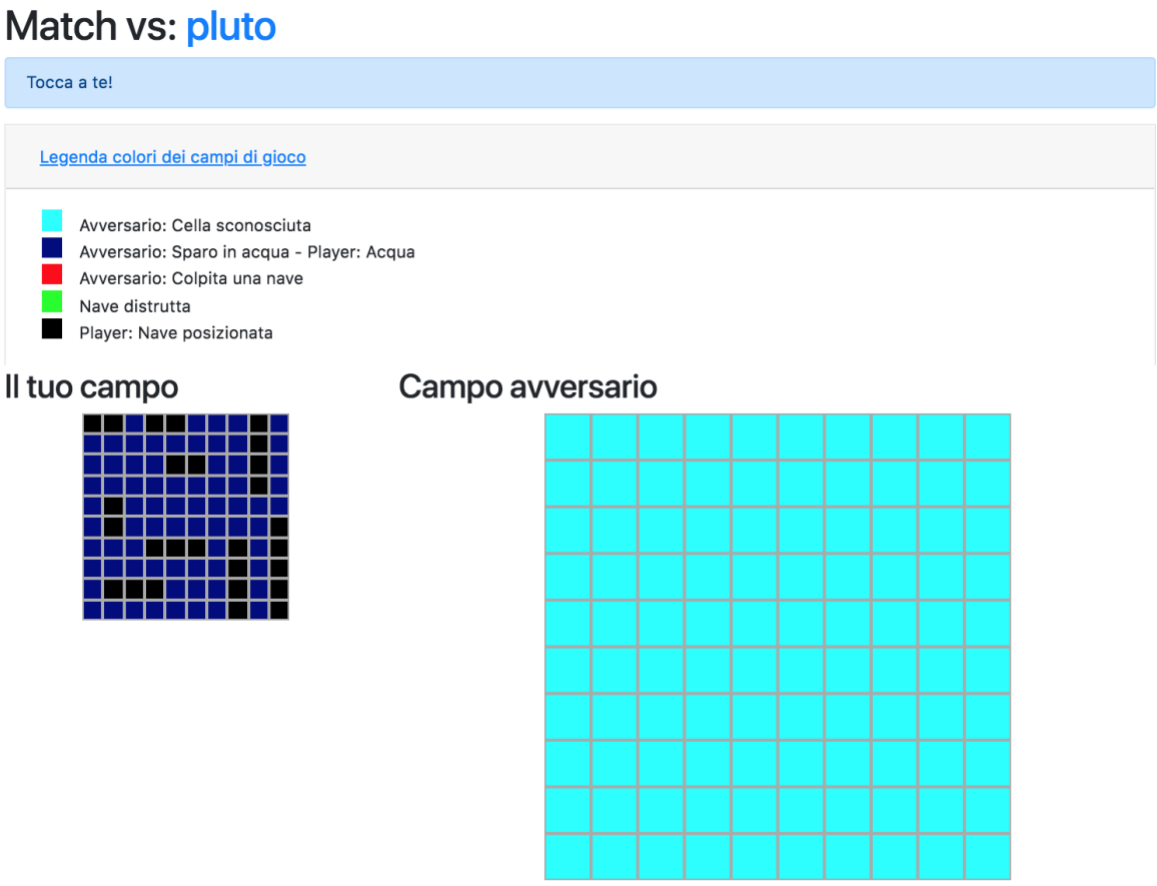


Figura 12: client 2

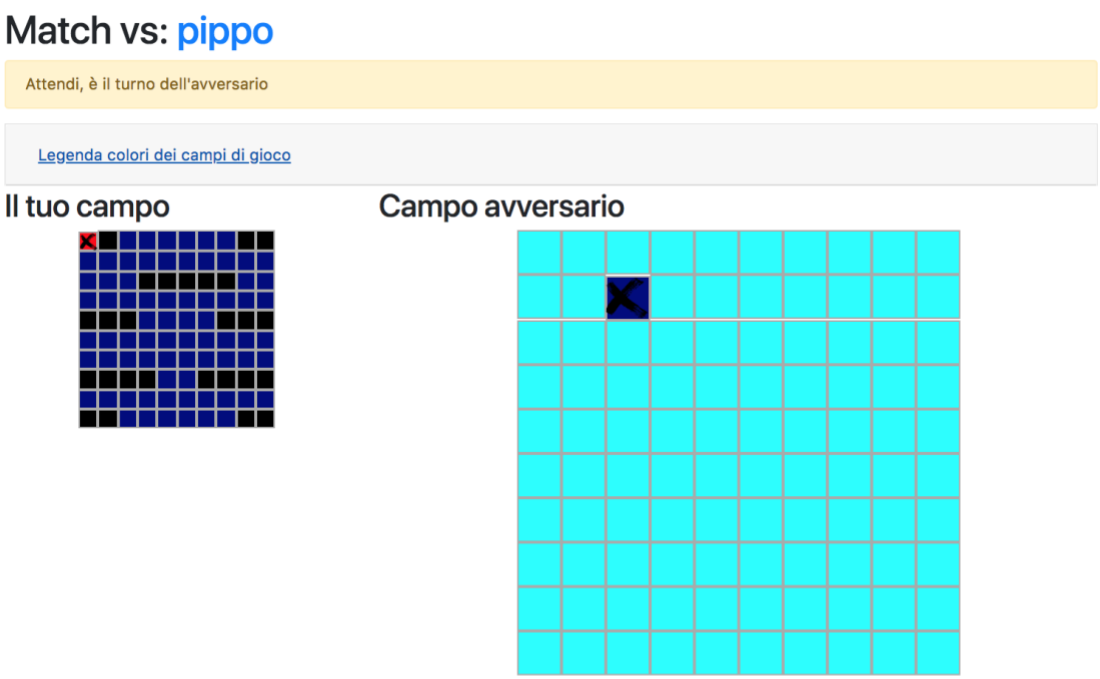


Figura 13: client 1

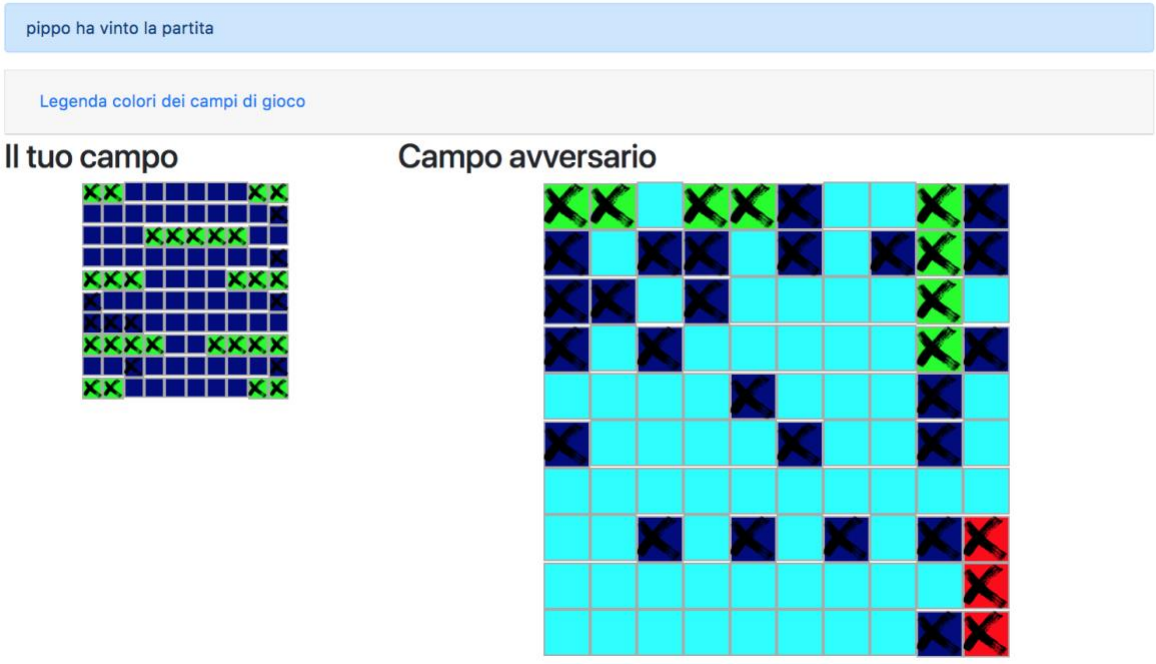


Figura 14: fine partita, client 1

Figura 15: fine partia, client 2

Valutazioni e considerazioni

Siamo a conoscenza che il software è abbastanza carente dal punto di vista della progettazione della UI e della UX, però il nostro focus in questo caso è stato quello di sviluppare realizzare in tempi ragionevoli un'applicazione che facesse uso del maggior numero di tecnologie per il web viste a lezioni. Non abbiamo ritenuto necessario l'utilizzo di REDIS all'interno dell'applicazione creata anche se sarebbe potuta essere una buona idea utilizzarlo come cache per le informazioni contenute in MongoDB