

Politecnico di Milano

Facoltà di Ingegneria Industriale e dell’Informazione

Corso di Laurea Triennale in Ingegneria Informatica

Immagine che contiene testo, Carattere, cerchio, schermata

Descrizione generata automaticamente

Candidato:

Nome: **Ettore Cirillo Mugisha**

Matricola: **991290**

Codice Persona: **10864944**

Relatore: Professore **Salnitri Mattia**

INDICE

[**1 - BLACK JACK SAFER** 3](#_Toc167312144)

[1.1 - Descizione della Tematica 3](#_Toc167312145)

[1.2 - Obiettivo 3](#_Toc167312146)

[1.3 - Analisi dei Requisiti 3](#_Toc167312147)

[1.3.1 - Requisiti Funzionali 3](#_Toc167312148)

[1.3.1.1 - Use Case Diagrams 4](#_Toc167312149)

[1.3.2 - Requisiti non funzionali 4](#_Toc167312150)

[1.3.3 - Design Strutturale 4](#_Toc167312151)

[**2 - Database** 5](#_Toc167312152)

[2.1 - Diagramma ER 5](#_Toc167312153)

[2.2 - Annotazioni Tecniche 5](#_Toc167312154)

[**3 - Back-End** 5](#_Toc167312155)

[3.1 - Progettazione Architetturale 5](#_Toc167312156)

[3 1.1 - Pattern MVCS 5](#_Toc167312157)

[**4 - Front-End** 5](#_Toc167312158)

[4.1 - Tecnologie Utilizzate 5](#_Toc167312159)

[4.1.1 - Angular 5](#_Toc167312160)

[4.1.1.1 - Componenti 6](#_Toc167312161)

[4.1.1.2 - Services 6](#_Toc167312162)

[4.1.1.3 Routes 6](#_Toc167312163)

[*4.1.1.4 Guards* 6](#_Toc167312164)

[4.1.2 – TailwindCss 7](#_Toc167312165)

[4.1.3 - Leaflet 7](#_Toc167312166)

[4.1.4 - Toastr 8](#_Toc167312167)

[4.2 - Progettazione Architetturale 8](#_Toc167312168)

[4.2.1 - Pattern MVVM 8](#_Toc167312169)

[4.2.2 - Foldering 9](#_Toc167312170)

[4.3 - Design Patterns 9](#_Toc167312171)

[4.3.1 – DTOs 9](#_Toc167312172)

[4.3.2 Dependency Injection & Singleton 9](#_Toc167312173)

[4.3.3 Observer 10](#_Toc167312174)

[4.4 - Login 10](#_Toc167312175)

[4.4.1 – Sequence Diagram 10](#_Toc167312176)

[4.5 – Partita a Black Jack 10](#_Toc167312177)

[4.5.1 – Sequence Diagram 10](#_Toc167312178)

[**5 – GIT** 11](#_Toc167312179)

[5.1 - Comandi Essenziali 11](#_Toc167312180)

[5.2 – Codice Sorgente 11](#_Toc167312181)

[**6 - Conclusioni** 11](#_Toc167312182)

# **1 - BLACK JACK SAFER**

## 1.1 - Descizione della Tematica

Il gioco d’azzardo, fin dai tempi più antichi, è sempre stato un’attività che ha caratterizzato l’esperienza umana, spesso utilizzato come valvola di sfogo per emozioni intense o come strumento per indurre eccitazione ed adrenalina nella vita quotidiana. Da esso nasce lo studio della probabilità e della statistica, disciplina di grande studio al giorno d’oggi.

Tra i giochi d’azzardo, il BlackJack si distingue per la sua intuitività ed accessibilità. Notoriamente riconosciuto per la possibilità di applicare strategie matematiche, il BlackJack è considerato da molti il gioco d’azzardo più “sicuro”. Tale percezione, deriva dalla capacità di ridurre il vantaggio del banco attraverso decisioni strategiche informate, basate su calcoli matematici esatti.

## 1.2 - Obiettivo

“BlackJack Safer” nasce con l’intento di trasformare l’approccio al gioco d’azzardo online, puntando alla riduzione dei rischi di dipendenza e all’educazione di essi alla strategia migliore. Questo obiettivo è perseguito attraverso una gestione più umana e responsabile del saldo degli utenti, con un sistema che prevede l’approvazione (o il rifiuto) per ogni operazione di ricarica. In questo modo, dunque, non solo si protegge l’utente dal rischio di comportamenti compulsivi, ma si propone anche di educarli ad un approccio al gioco più consapevole e controllato.

Il software utilizza tecnologie moderne e strategie di intervento precoce per garantire un’esperienza di gioco equilibrata e sicura, mirando a mantenere il divertimento del BlackJack senza trascurare la salute ed il benessere dei giocatori.

## 1.3 - Analisi dei Requisiti

### 1.3.1 - Requisiti Funzionali

I requisiti funzionali rivestono un ruolo cruciale nel processo di sviluppo del software, in quanto costituiscono un pilastro per la progettazione, lo sviluppo, il controllo e la convalida. Documentare i requisiti funzionali contribuisce a garantire che il software risponda alle esigenze degli utenti e dell'organizzazione. Il progetto prevede l'interazione tra tre differenti tipi di utenti:

**ADMIN –** Agirà come supervisore dell’applicazione. Esso avrà la possibilità di: creare/Eliminare dei “Tabacchi” dove gli utenti potranno richiedere di ricaricare il saldo, Creare/Modificare/Eliminare gli Economi, un particolare tipo di utente i quali gestiranno le ricariche di denaro, Modificare/Eliminare gli utenti PLAYER.

**ECONOMO –** Agirà come gestore del saldo degli utenti Player. Esso, associato ad uno o più specifici Tabacchi, avrà la possibilità di accettare o rifiutare le richieste di ricarica mandate dagli utenti. L’esito della richiesta sarà notificato successivamente al mittente.

**PLAYER –** Agirà come giocatore effettivo del casinò. Esso avrà la possibilità di unirsi ad un tavolo e giocare la sua partita a blackJack fin quando vuole. Qualora occorre, avrà la possibilità di effettuare una richiesta di ricarica del saldo, la quale dovrà essere approvata dall’utente Economo.

#### 1.3.1.1 - Use Case Diagrams

<Metti i diagrammi con immagini>

### 1.3.2 - Requisiti non funzionali

I requisiti non funzionali di una Web Application, se rispettano gli standard e le normative del settore, sono essenziali per garantire una user experience migliore possibile. Esse si focalizzano sulle performance, sulla qualità e la sicurezza. In questo progetto sono rese in evidenza le più rilevanti tra cui:

**Scalabilità –** Uno dei requisiti più importanti del software è quello di avere una struttura facile da mantenere, aggiornare ed estendere in futuro. Essa, infatti, pone le fondamenta per una manutenibilità più agevole e conveniente per gli sviluppatori futuri. Tra gli aspetti più importanti della scalabilità del software troviamo:

* **Leggibilità –** Scrivere il codice secondo le convezioni globali e rispettando le norme della programmazione è una buona prassi per rendere l’applicazione facile da capire per gli sviluppatori futuri.
* **Commenti e Documentazione -** I commenti, con anche l’uso della documentazione, velocizzano notevolmente il reverse engineering degli sviluppatori.
* **Versionamento –** L’uso di tool di controllo di versione del codice, come *Git*, permette agli sviluppatori di tenere traccia delle modifiche implementate sul software, potendo così navigare tra esse in maniera facile.
* **Testing –** Esso è fondamentale per verificare che l’applicazione funzioni correttamente, potendo così identificare casi limite e bug di sistema.

**Sicurezza –** pietra miliare nel mondo dell’IoT, essa è fondamentale per la protezione da minacce, pericoli o rischi che comprometterebbero l’integrità e la disponibilità dei dati e delle informazioni, come i dati sensibili degli utenti. Tra gli aspetti chiave della sicurezza troviamo:

* **Integrità –** Essa comprende la crittografia dei dati, l’implementazione di controlli di accesso e validazione dei dati in ingresso volti a garantire che i dati o il sistema non risultino alterati in modo anomalo.
* **Autenticazione e autorizzazione –** Per verificare l'identità di un utente e consentire l'accesso a risorse o funzionalità specifiche, utilizziamo il *JWT* (JSON Web Token). Questo token crittografato contiene dati sull'utente autenticato e può essere validato per assicurarne l'autenticità. Questo metodo fornisce un modo efficiente e sicuro per gestire l'accesso degli utenti, permettendo loro di accedere a determinate parti dell'applicazione solo se sono correttamente autenticati e autorizzati.

**Applicabilità –** Essa misura la facilità d’utilizzo con la quale l’utente prossimo riesce ad interagire con l’applicativo in modo efficace. Tra i vari aspetti chiave troviamo:

* **UX/UI:** L’interfaccia grafica dell’applicazione deve essere facile ed intuitiva per garantire che l’utente non abbia difficoltà a navigare nell’applicazione

### 1.3.3 - Design Strutturale

L’applicazione si struttura su 3 livelli; tale approccio, tipico della progettazione di Web Application, mira a suddividere l’applicazione in tre componenti distinti per garantire una disposizione ben organizzata e scalabile. Ogni livello svolge un ruolo preciso e si occupa di determinati aspetti all’interno dell’applicazione.

15 fundamental tips on REST API design | by Williams O | MediumImmagine che contiene testo, schermata, logo, automobile

Descrizione generata automaticamenteImmagine che contiene Elementi grafici, grafica, clipart, creatività

Descrizione generata automaticamenteImmagine che contiene simbolo

Descrizione generata automaticamente



**Client** – Esso avrà il compito di mostrare all’utente le informazioni da lui richieste in modo intuitivo, ottenute tramite richieste al server.

**Server** – Esso avrà il compito di fornire i servizi o le risorse richieste dal Client, avendo interazione con il Database.

**Database** – Esso rappresenta l’effettivo sistema di archiviazione dell’applicazione. Esso ha la possibilità di organizzare, memorizzare e recuperare dati in maniera dinamica.

# **2 - Database**

## 2.1 - Diagramma ER

<Metti l’immagine del giagramma ER>

## 2.2 - Annotazioni Tecniche

La scelta di utilizzare un *relational database management system* come PostreSQL è stata fatta per varie ragioni: poiché open source, esso ha alle spalle una comunità di sviluppatori che contribuiscono a mantenerlo; inoltre, esso possiede funzionalità performanti per quanto riguarda la **sicurezza** e la **geolocalizzazione**.

Per la tabella dell’utente, è stato scelto il nome “***\_user***” poiché “*user*” è una parola riservata in diversi DBMS, tra cui *Postgres*.

<Capire se mettere l’enum>

# **3 - Back-End**

## 3.1 - Progettazione Architetturale

### 3 1.1 - Pattern MVCS

In questo contesto, è stato implementato il pattern architetturale MVCS (Model-View-Controller-Service), una variante del pattern MVC. Utilizzato per applicazioni complesse, esso ha ho scopo di organizzare e separare le responsabilità di ogni componente, consentendo una migliore manutenibilità e scalabilità del codice. Se ne illustrino i componenti:

Model – Esso contiene le entità di dominio, mappati sul Database, e i metodi di accesso ai dati (rappreentati dai repositories).

View – Esso è il responsabile dei dati della User Interface dell’applicazione, costituito principalmente dall’output generato dagli endpoint.

Controller – Esso rappresenta l’interfaccia di accesso all’applicazione. Ha il compito di intercettare le richieste http da parte del client e indirizzano la richiesta al service dedicato.

Service – Esso è il vero responsabile della business logic. Gestisce la logica dell’applicazione, alleggerendo lavoro ai controller.

### 3.1.2 Packaging

<inserire immagine del packaging>

## 3.2 Exception

Durante l’utilizzo dell’applicazione, può accadere che una richiesta http generi anomalie o errori. Se ciò accade, l’applicazione deve essere in grado di individuare e catchare gli errori e notificare il client di quanto è accaduto.

In questo contesto, si è deciso di implementare un sistema di gestione delle eccezzioni, composto da un’entità con messaggio e status come attributi, un controller responsabile dell’invio della risposta al client e un insieme di classi children della classe RuntimeException sviluppate secondo il tipo di errore rilevato.

### 3.2.1 Class Diagram

<Mettere il class Diagram>

## 3.3 Dominio

Come detto in precedenza,

# **4 - Front-End**

## 4.1 - Tecnologie Utilizzate

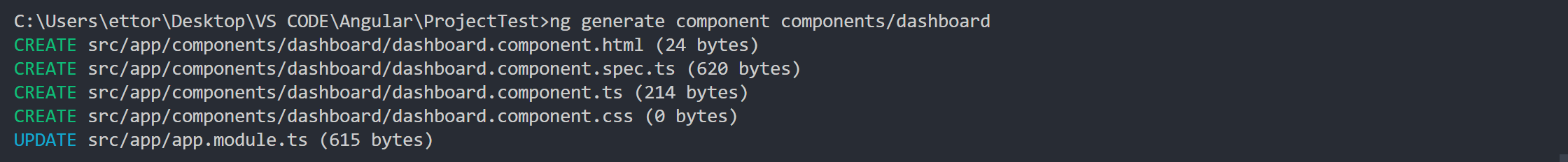
### 4.1.1 - Angular

Angular è un framework **open-source** sviluppato per la maggior parte da *Google*. Esso è implementato con *TypeScript*, utilizzando un approccio modulare che può **OOP** (Object Oriented Programming). La creazione di tali **componenti**, dunque, rende le Web Application dinamiche e scalabili.

Tra le caratteristiche principali di Angular troviamo la gestione del **routing**, l’iniezione delle **dipendenze**, la manipolazione avanzata del *DOM*, la gestione agile degli eventi e un sistema di **binding dei dati bidirezionale**.

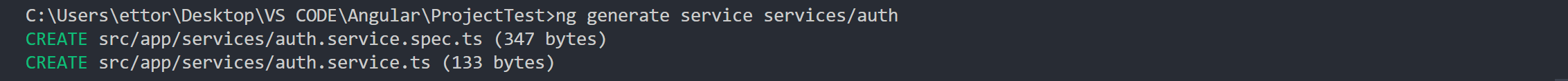
#### 4.1.1.1 - Componenti

Essi rappresentano dei blocchi che mostrano la suddivisione dell’applicazione. Sono composti da una struttura HTML, una logica di programmazione e i metadati per formare elementi modulari e riutilizzabili nelle interfacce utente delle applicazioni web. Questo approccio mira a migliorare la scalabilità del codice, utilizzando un approccio che agevola l’implementazione del sistema nel tempo.

*Creazione di un componente tramite Angular CLI. Se non esiste già, verrà generata automaticamente una cartella di nome “components” con dentro un’altra cartella “login” con dentro i file mostrati in figura.*

#### 4.1.1.2 - Services

Essi sono delle classi fondamentali per la condivisione di funzionalità tra i vari componenti all’interno dell’applicazione, inoltre, sono responsabili della gestione della comunicazione con il back-end. Difatti, grazie alla loro iniezione nei componenti, i servizi favoriscono la modularità e la scalabilità del codice, consentendo una gestione più efficiente e snodata delle diverse parti dell’applicazione

*Creazione di un service tramite Angular CLI. Se non esiste già, verrà generata automaticamente una cartella di nome “services” con dentro i file mostrati in figura**.*

#### 4.1.1.3 Routes

Essi sono strumenti fondamentali per la gestione della navigazione all’interno di una ***Single Page Application (SPA).*** La loro funzionalità è quella di definire percorsi specifici all'interno dell'applicazione e associare ognuno di essi a un componente specifico. Questi percorsi vengono poi riflessi nella barra degli Url del browser durante la navigazione dell’utente.

L’interazione tra percorsi e componenti semplifica la gestione della navigazione all’interno della *SPA*, poiché consiste di identificare in modo chiaro quale componente deve essere caricato per ciascun percorso, consentendo una navigazione fluida e intuitiva all'interno *dell'applicazione*.

*Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteVengono definite due rotte: “registration” e “dashboard”, ognuna associata ad un componente specifico.*

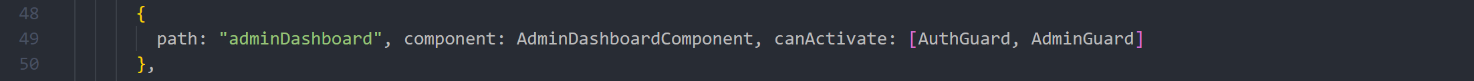
*4.1.1.4 Guards*

Le guards sono meccanismi di sicurezza essenziali in Angular che contribuiscono a proteggere le route dell'applicazione. Una guard viene utilizzata per decidere se un utente può accedere o meno a determinate parti dell'applicazione, basandosi su condizioni specifiche come l'autenticazione e i permessi dell'utente. Questo processo di verifica è cruciale per la gestione dell'accesso e per garantire che risorse sensibili siano esposte solo a utenti autorizzati. Le guards possono anche gestire reindirizzamenti automatici, ad esempio rimandando gli utenti non autenticati alla pagina di login.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

*Esempio di auth.guard che invoca la funzione nello user.service, la quale controlla se l’utente è autenticato, qualora non lo fosse, viene reindirizzato sulla pagina di login.*

*Esempio di applicazione di 2 guard sulla rotta adminDashboard, queste 2 Guards avranno il compito di verificare se l’utente è autenticato ed ha il ruolo “ADMIN”.*

### 4.1.2 – TailwindCss

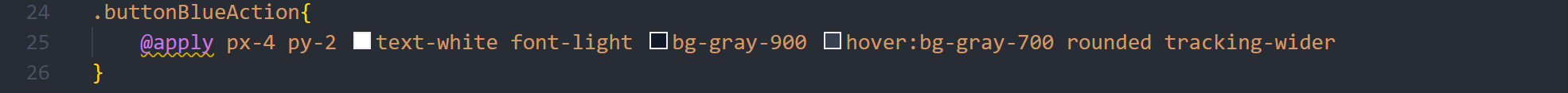
*TailwindCss* è un framework *CSS* progettato per agevolare lo sviluppo e la manutenzione del codice *CSS*. Esso si distingue dagli altri framework del settore per la sua fornitura di classi di utilità CSS **basate su singole proprietà**. Tra le sue caratteristiche troviamo:

**Leggibilità –** Le classi di utilità sono definite in modo intuitivo, aumentando quindi la comprensione del codice.

*Esempio di utilizzo di una classe Tailwind in un elemento HTML.*

**Basso livello di astrazione –** Tailwind si trova ad un livello molto basso di astrazione, consentendo un controllo più dettagliato e diretto degli stili CSS applicati.

**Elasticità –** È possibile personalizzare il set di classi nel file di configurazione per adattarlo a esigenze specifiche.

*Esempio di creazione di una classe buttonBlueAction dove, tramite la keyword* ***@apply****, vengono applicate classi Tailwind già esistenti.*

### 4.1.3 - Leaflet

Libreria TypeScript *open-source*, essa fornisce un insieme di strumenti per la creazione di mappe interattive all’interno di Web Applications. Tra le sue peculiarità troviamo:

**Legerezza** – Leaflet è notoriamente leggero, associato ad un file *TypeScript* di dimensioni ridotte. Questa accoppiata contribuisce a ridurre i tempi di caricamento delle mappe integrate.

**Estendibilità** – La libreria è estendibile attraverso l’uso di *plugin,* che permettono di aggiungere funzionalità specifiche alle mappe.

*Struttura HTML per la visualizzazione della mappa.*

*Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteMetodo di configurazione base per una mappa generica.*

### 4.1.4 - Toastr

Essa è una libreria utilizzata per visualizzare un particolare tipo di notifiche, chiamate **notifiche toast**, sulla pagina web. Le notifiche toast sono piccoli messagi che compaiono per un tempo determinato nella parte superiore dello schermo per informare gli utenti di eventuali avvisi o eventi **senza interrompere il flusso di esecuzione dell’applicazione.**

*Comando per l’installazione della libreria*

*Immagine che contiene schermata, testo, Software multimediale

Descrizione generata automaticamenteConfigurazione all’interno dell’app.module dell’applicazione.*

*Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteTesting metodi offerti dalla libreria.*



*Output visivo del codice scritto prima alla chiamata del metodo* ***testToastr().***

## 4.2 - Progettazione Architetturale

### 4.2.1 - Pattern MVVM

Nelle Web Application sviluppate in *Angular*, è buona norma utilizzare il pattern architetturale **Model View View-Model (MVVM).** Esso viene utilizzato per dividere la logica di presentazione dall’implementazione sottostante, offrendo notevoli vantaggi quali: **legame bidirezionale dei dati**, **aggiornamento automatico dell’interfaccia Grafica**, **riutilizzabilità e scalabilità del codice**.

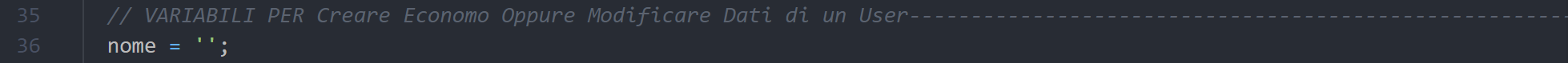
Come detto in precedenza, i componenti del patter sono:

**Model –** Esso rappresenta i dati dell’applicazione recuperati dalle ***RESTful API***, chiamate dai service.

**View –** Esso è composto da ***template HTML***. Ha il compito di rappresentare i dati del model nell’interfaccia grafica. La separazione della componente grafica contente di mantenere la logica di presentazione divisa dalla logica di business, come da buona norma.

**View-Model –** I componenti precedentemente espressi rappresentano i View-Model, poiché in essi è contenuta la logica che connette il modello alla vista. Grazie al **two-way data binding** e le direttive di Angular, i componenti tengono sincronizzati il modello e la vista.

*Esempio di two-way data binding. Attraverso la keyword ngModel di Angular si sincronizza la variabile dichiarata nell’apposito componente.*

*Dichiarazione ed inizializzazione della variabile con lo stesso nome presente nell’ngModel. <chiarire se mettere il protected>*

### 4.2.2 - Foldering

<immagine del foldering>

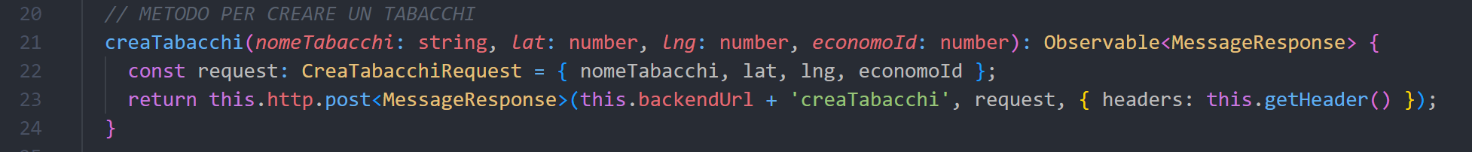
## 4.3 - Design Patterns

### 4.3.1 – DTOs

Il pattern ***DTO*** *(Data Transfer Object)* è utilizzato per trasferire dati tra componenti di un’applicazione tramite oggetti specifici. Tali oggetti contengono solamente dati, senza includere business logic. Questo design pattern mira ad agevolare la comunicazione e lo scambio di dati all’interno dell’applicazione. In questo contesto, come da buona norma, all’interno dell’apposita cartella sono presenti due sottocartelle per dividere i vari oggetti in base al tipo: ***request*** o ***response***.

*Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteDichiarazione tramite interfaccia TypeScript del DTO per la richiesta della creazione di un tabacchi con i tipi di dati richiesti.*

*Esempio di applicazione di DTO request e response. In questo caso, viene chiamato il backEnd per eseguire la creazione di un tabacchi con i dati richiesti.*

### 4.3.2 Dependency Injection & Singleton

La ***DI (Dependency Injection)*** è un pattern indispensabile in *Angular.* Ecco consiste nell’assegnare alle classi le dependency che le necessitano durante l’inizializzazione. Qui di seguito vengono illustrati i passaggi per applicare efficacemente il pattern:

**Provider –** Le dependency sono definite come *provider.* Un *provider* è un oggetto che ha il compito di comunicare al sistema di *DI* come creare o ottenere una dependency.

**Iniezione –** Le dependency vengono iniettate nei costruttori delle classi di cui ne hanno bisogno.

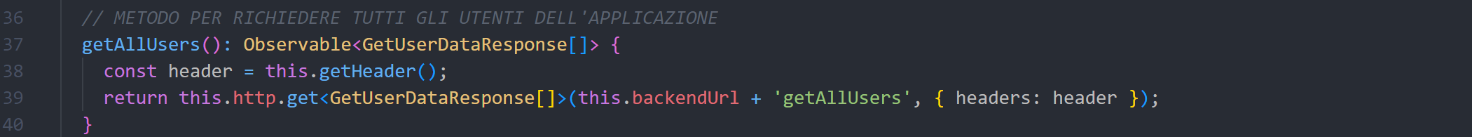
**Configurazione –** I *provider* sono configurati a livello di componenti per l’annotazione **@Injectable**. L’attributo ***providedIn: “root”***, il quale inietta il servizio a livello globale, rappresenta implicitamente l’uso in **singleton**, creando una sola istanza di questi provider condivisa con tutta l’applicazione.

*Immagine che contiene testo, schermata

Descrizione generata automaticamenteEsempio di DI. Il modulo per le richieste http viene iniettato in un service per comunicare con il server.*

### 4.3.3 Observer

Esso è un modello di **programmazione asincrona**, noto per gestire sequenze di dati o avvenimenti in modo dinamico per le richieste *HTTP* al server. Questo tipo di pattern fornisce una gestione più efficace rispetto alla solita classe *Promise* di *TypeScript*. Il pattern si può costruire in 2 semplici passaggi che normalmente sono implementati nei *services* e nei *components*. Il primo si occupa di interfacciarsi con le classi esterne, mentre il secondo comunica il service attraversi l’***”iscrizione”*** dell’evento apposito.

*Dichiarazione di una richiesta HTTP. In questo caso vengono richiesti tutti gli utenti presenti nel Database.*

*Immagine che contiene testo, schermata, software

Descrizione generata automaticamenteTipica gestione delle richieste HTTP da parte dei components. Essi si iscrivono all’evento ed attendono una risposta che verrà successivamente gestita nell’apposita* ***arrow function => {.***

## 4.4 - Login

Bla bla

### 4.4.1 – Sequence Diagram

<Mettici l’immagine del sequence>

## 4.5 – Partita a Black Jack

Blablabla

### 4.5.1 – Sequence Diagram

<Metti l’immagine del sequence>

# **5 – GIT**

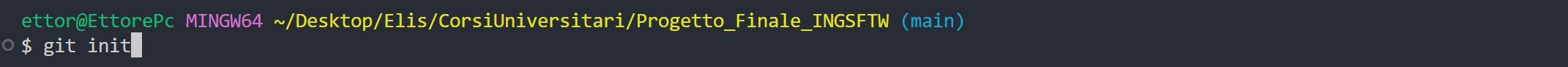
**Git** è un **sistema di controllo di versione** ampliamente utilizzato per gestire progetti software e tener traccia dei cambiamenti al codice sorgente nel tempo. Esso risulta molto comodo agli sviluppatori poiché consente loro di collaborare in maniera efficace e mantenere uno storico delle modifiche apportate al codice da loro stessi. Tra i vari componenti fondamentali troviamo:

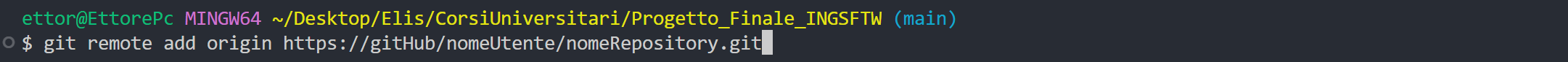
**Repository –** Spesso abbreviata con *repo*, essa rappresenta un archivio che contiene tutti i file, le cartelle e la cronologia delle modifiche di un progetto. Essa può essere **locale** (sul computer) o **remota** (su un server).

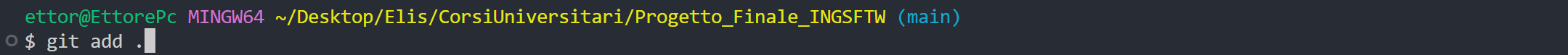
**Branching –** Esso rappresenta un “*ramo*”, ovvero una copia separata del codice che permette di lavorare su nuove features senza compromettere il ramo principale.

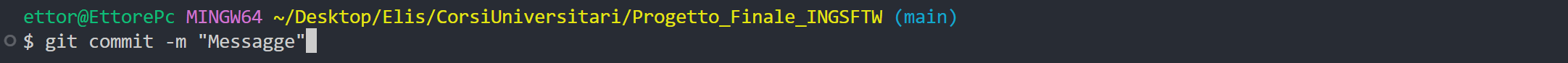
**GitHub/GitLab –** Servizi che offrono hosting remoto per le repositories *Git*. Essi forniscono funzionalità di collaborazione, richieste, clonaggio etc…

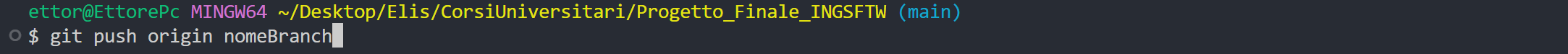
## 5.1 - Comandi Essenziali

 *Inizializzazione di una nuova repository nella directory corrente.*

 *Collegamento della repository locale alla repository remota, raggiunta tramite il link inserito.*

 *Aggiunta di tutti i files alla repository locale.*

 *Snapshot delle modifiche apportate ai file con associato un messaggio.*

 *Caricamento del commit locale nella repository remota.*

Il primi 2 comandi vengono utilizzati solamente alla creazione del progetto, invece gli utltimi 3, in sequenza, ogni volta si ritiene necessario notificare modifiche al progetto.

## 5.2 – Codice Sorgente

Questa applicazione è open-source, e come tale, è disponibile al seguente link:

https://github.com/EccirilloM/BlackJack

# **6 - Conclusioni**