UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

PROGETTAZIONE E REALIZZAZIONE MEDIANTE FRAMEWORK ANGULAR E SPRING DI UN NFT STORE

Relatore:
Prof. Matteo Comin

Laureando:
Tommaso Michelon
matricola: 1216360

22/07/2022

ANNO ACCADEMICO 2021 – 2022

SOMMARIO

Questo documento presenta il lavoro svolto dal laureando Tommaso Michelon durante il periodo di tirocinio dal 14/02/2022 al 22/04/2022, della durata di circa duecentocinquanta ore, presso l'azienda Sync Lab S.r.l.

La prima parte del tirocinio consisteva in formazione generale sullo sviluppo web e più nello specifico su due framework: Spring per il back-end e Angular per il front-end.

La seconda parte, invece, prevedeva la progettazione e la realizzazione di un applicativo per la gestione degli NFT da parte di utenti registrati, che potevano aggiungere o vendere i loro token come in un vero e proprio NFT store.

INDICE

1. **INTRODUZIONE** 11
1.1 L'azienda 11
1.2 Obiettivi del tirocinio 12
1.3 Formazione iniziale 13
1.3.1 Metodologia AGILE 13
1.3.2 Principi SOLID 15
1.3.3 Sviluppo Web 16
1.3.4 Servizi REST 17

2. **BACK-END** 19
2.1 Java SE 19
2.2 Spring 20
2.2.1 Spring Boot 21
2.2.2 Spring Data JPA 21
2.2.3 Spring Data REST 22
2.3 Strumenti ausiliari 22
2.3.1 Maven 22
2.3.2 Postman 23
2.3.3 Git 23

3. **FRONT-END** 25
3.1 TypeScript 25
3.2 Angular 26
3.2.1 Single Page Application 27
3.2.2 Design Pattern 28
3.2.3 Angular CLI 28
3.3 Bootstrap 28
3.4 Node.js 29
3.5 Npm 29

4. **PROGETTO FINALE: NFT STORE** 31
4.1 Descrizione progetto 31
4.2 Rappresentazione dati su database 31
4.3 Funzionalità sviluppate – Gestione Account 32
4.3.1 Registrazione 32
4.3.2 Login 33
4.3.3 Logout 33
4.3.4 Modifica password 34
4.3.5 Eliminazione account 35
4.4 Funzionalità sviluppate – Gestione NFT 36
4.4.1 Visualizzazione NFT 37
4.4.2 Aggiunta NFT 37
4.4.3 Vendita NFT 38

5. **CONCLUSIONI** 41

6. **BIBLIOGRAFIA** 43

Capitolo 1

INTRODUZIONE

In questo capitolo verranno descritte l'azienda ospitante, gli obiettivi del tirocinio e i temi trattati durante la formazione iniziale.

Figura 1.1: logo azienda Sync Lab

1.1 L'azienda

Sync Lab S.r.l. è una società di consulenza informatica fondata nel 2002 con sedi a Napoli, Roma, Milano, Padova, Verona e Como.

Potendo contare sull'esperienza e le competenze dei suoi oltre 300 dipendenti, realizza prodotti e soluzioni per diversi mercati quali: Sanità, Industria, Energia, Telco, Finanza e Trasporti & Logistica.

Numerose sono le certificazioni che distinguono la qualità del lavoro offerto dall'azienda, quali: ISO 9001, ISO 14001, ISO 27001 e ISO 45001.

Attraverso il laboratorio di ricerca e sviluppo riesce ad offrire prodotti software e soluzioni sempre all'avanguardia e in linea con le nuove tendenze e le sfide che emergono in numerosi campi del mondo ICT, come: GDPR, Big Data, Cloud Computing, IoT, Mobile e Cyber Security.

1.2 Obiettivi del tirocinio

È possibile suddividere gli obiettivi del tirocinio in due categorie: obiettivi personali e aziendali.
Gli obiettivi personali riguardavano sia l'apprendimento e la conoscenza degli strumenti utilizzati, in ambito lavorativo, per la progettazione di applicativi Web, sia tutto ciò che comprendeva l'ingresso in una realtà aziendale operante nel settore dell'informatica.
Più nello specifico, l'interesse principale era focalizzato sul back-end e sui framework utilizzati per implementarlo, ma anche un approccio al front-end per capire, in maniera più globale, il funzionamento completo di un'applicazione progettata per il Web.
Oltre alla formazione però, riuscire a capire come fosse organizzata un'azienda, come lavorassero i vari team su progetti affidati dai clienti e come i singoli dipendenti affrontassero le loro attività quotidiane, è stato un aspetto con un notevole peso nella scelta di affrontare un percorso di tirocinio.

Per quello che riguarda gli obiettivi aziendali, invece, l'interesse riguardava perlopiù l'approccio dello studente a una nuova realtà e alle sfide e alle difficoltà nel portare avanti un progetto in un contesto completamente differente da quello offerto dall'università.
Come esplicato dal tutor aziendale, Sync Lab dedicava molte risorse alla formazione di studenti, su differenti temi e argomenti, con lo scopo finale di una possibile assunzione all'interno dell'azienda.

1.3 Formazione iniziale

In questa sezione verranno descritti gli argomenti principali affrontati durante la formazione aziendale: la metodologia *agile* e i principi *solid*, applicati all'interno dell'azienda, un'introduzione generale allo sviluppo web e i servizi *rest*, utilizzati per lo sviluppo del progetto finale.

1.3.1 Metodologia AGILE

La metodologia di sviluppo software *AGILE* è basata sul rilascio di distribuzioni efficienti, create in modo rapido e tramite la collaborazione per garantire un miglioramento continuo. Solitamente i gruppi di lavoro sono costituiti da pochi sviluppatori in grado di organizzarsi in autonomia, ma un'importanza strategica è attribuita agli incontri periodici fra team per garantire una condivisione d'intenti e di obiettivi.

A differenze dei modelli precedenti di ingegneria del software, basati su fasi precise e ben delineate e su uno sviluppo sequenziale del prodotto, la metodologia *agile* permette maggior flessibilità sia nella raccolta di specifiche, adeguabili anche durante l'avanzamento dello sviluppo software, sia nell'implementazione e nei test del codice prodotto.

I valori su cui si basa questa metodologia sono dichiarati nel *Manifesto Agile* e sono quattro:

- Gli individui e le interazioni più che i processi e gli strumenti
- Il software funzionante più che la documentazione esaustiva
- La collaborazione col cliente più che la negoziazione dei contratti
- Rispondere al cambiamento più che seguire un piano

Allo stesso tempo, il *Manifesto Agile* non intende fornire regole da seguire in maniera assoluta, ma piuttosto delle linee guida da seguire e da adattare alle proprie necessità.

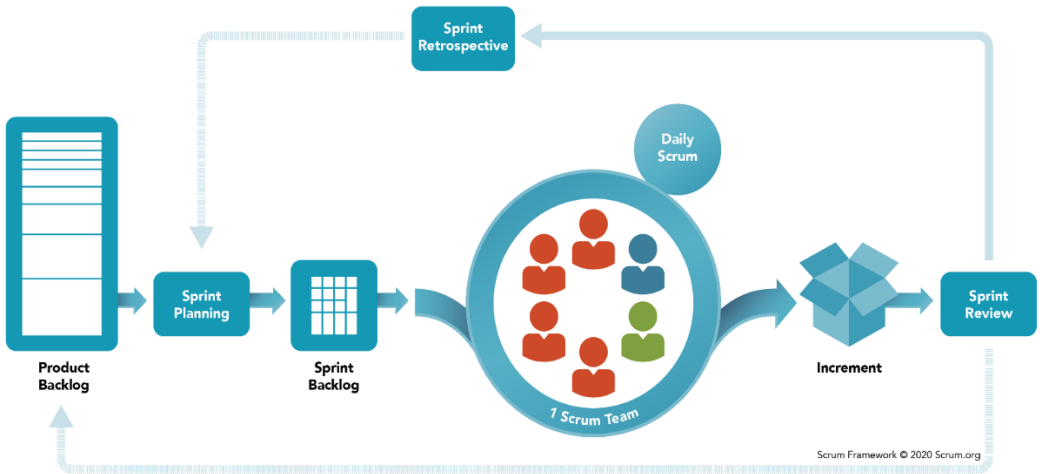

Figura 1.2: schema funzionamento Scrum

Un esempio concreto di metodologia *agile* è il framework di processo *SCRUM*, una vera e propria modalità strutturata e pianificata con ruoli, *sprint*, eventi e artefatti.

- Ruoli: negli *Scrum Team* sono presenti tre responsabilità distinte: il *Product Owner*, che deve garantire che i requisiti di prodotto siano centrati sui bisogni dei clienti, lo *Scrum Master*, che è responsabile della corretta esecuzione del processo e i *Developer*, che sono coloro che realizzano il lavoro effettivo.
- Sprint: è l'unità di base dello sviluppo, detto anche ciclo, che può durare fino a quattro settimane. Al termine di ogni iterazione il team rilascia uno o più incrementi, definiti in precedenza.
- Eventi: sono definiti per garantire trasparenza e ispezione del lavoro svolto. All'inizio di ogni sprint c'è lo *Sprint Planning* durante il quale il team seleziona le attività da svolgere e completare entro il termine del ciclo. Ogni giorno si tiene il *Daily Scrum* per verificare cosa è stato sviluppato, con cosa si procederà ora e se sono sorti problemi. Al termine dello sprint avvengono due eventi: lo *Sprint Review*, per ispezionare il lavoro svolto assieme ai clienti, e lo *Sprint Retrospective*, momento dedicato al team per riflettere su come migliorare in futuro.
- Artefatti: rappresentano, in linea generale, il lavoro e il valore in diversi modi: il *Product Backlog*, è una lista dei requisiti di un prodotto con associata una priorità e rappresenta il lavoro da svolgere, lo *Sprint Backlog*, è uno schema di ciò che deve essere svolto durante uno sprint con le varie fasi di avanzamento, e il *Burn Down*, che è una rappresentazione grafica del lavoro da svolgere in un determinato periodo di tempo.

1.3.2 Principi SOLID

I principi *SOLID* sono un paradigma che riassume le migliori pratiche per la programmazione orientata agli oggetti, secondo *Robert C. Martin*, per ridurre la complessità che tende a emergere col susseguirsi di richieste di modifica a un'applicazione software.
Il termine *SOLID* è l'acronimo usato per rappresentare i cinque principi:

- *Single responsability*: ogni classe dovrebbe avere una ed una sola responsabilità, interamente incapsulata al suo interno. Con responsabilità si intende un motivo per cambiare e quindi una classe dovrebbe avere uno, e uno solo, motivo di cambiamento.
- *Open-closed*: un oggetto o un'entità dovrebbe essere aperta alle estensioni, ma chiusa alle modifiche. Questo implica che dopo aver terminato l'implementazione di un'entità, questa non dovrebbe più subire modifiche al codice sorgente.
- *Liskov substitution*: gli oggetti dovrebbero poter essere sostituiti con dei loro sottotipi, senza alterare il comportamento del programma che li utilizza.
- *Interface segregation*: è preferibile che le interfacce siano molte, specifiche e piccole piuttosto che poche, generali e grandi.
- *Dependecy inversion*: i moduli di alto livello non devono dipendere da quelli di basso livello ed entrambi devono dipendere da astrazioni. È preferibile che le componenti di alto livello e di basso livello dipendano da astrazioni, ma mentre quelle di alto livello usano determinate astrazioni, quelle di basso livello le implementano. Questo per evitare dipendenze di compilazione.

I principi *SOLID* sono da intendersi come linee guida per lo sviluppo di software leggibile, estendibile e mantenibile, in particolare nel contesto di pratiche di sviluppo agili.

1.3.3 Sviluppo Web

Lo sviluppo web ha vissuto negli anni un profondo cambiamento, dovuto dalla continua evoluzione delle architetture e tecnologie e soprattutto dalle esigenze degli utenti.
Se un tempo le pagine web erano un insieme di semplici campi di testo, oggi in primo piano c'è una presentazione elaborata di contenuti multimediali, gestita attraverso pagine web dinamiche, dove l'aspetto grafico gioca un ruolo importante per rendere l'esperienza utente il più intuitiva possibile.
Per rendere tutto ciò possibile è necessario affidarsi alla programmazione web che, tramite diverse tecniche e linguaggi, permette la realizzazione e lo sviluppo di applicazioni per il Web.
L'architettura su cui si basa lo sviluppo web è detta *client/server*, dove il *client* si connette ad un *server* per la fruizione di un determinato servizio, come la ricezione di risorse o di informazioni.
Il *client* può essere rappresentato come un'interfaccia per inviare richieste al *server*, in maniera diretta o tramite nodi intermediari, grazie ai metodi offerti dai numerosi protocolli di rete.
Il *server* si occupa della gestione degli accessi, dell'allocazione e del rilascio delle risorse, della condivisione dei dati e della sicurezza. Quest'ultimo aspetto sta assumendo un ruolo sempre più centrale e indispensabile per garantire la protezione dei dati e delle connessioni.

Per quanto riguarda la programmazione web invece, è possibile suddividerla in programmazione *lato client*, comunemente conosciuto come *front-end*, e *lato server*, conosciuto come *back-end*.
Il *front-end* si occupa di tutto ciò che l'utente vede e con cui interagisce, ad esempio interfacce per l'acquisizione di dati o per la visualizzazione di contenuti multimediali. I linguaggi più diffusi sono: JavaScript, TypeScript, AJAX…
Mentre il *back-end* si occupa della gestione e dell'elaborazione dei dati raccolti dal *front-end*, grazie all'integrazione con i database e con i protocolli di cui necessita. I linguaggi più diffusi sono: Java, Python, PHP, .NET, ASP…

1.3.4 Servizi REST

Il modello architetturale *REST* definisce un insieme di principi che pongono al centro il protocollo HTTP (HyperText Transfer Protocol) e il concetto di risorsa identifica tramite URI (Uniforme Resource Identifier).
Entrando nello specifico, deve essere presente un'architettura client/server con richieste gestite tramite i metodi di HTTP per lo scambio di informazioni e risorse, inoltre la comunicazione dev'essere *stateless*, che quindi non prevedendo la memorizzazione delle richieste GET del client e rendendo perciò ogni richiesta distinta e non connessa alle precedenti.
Un altro concetto importante riguarda la presenza della memoria *cache* sia *lato client*, salvando le risposte ricevute, che *lato server* per ottimizzare le richieste future; inoltre, il fatto che la struttura del sistema debba essere "a strati", non presupponendo che le applicazioni client e server si connettano direttamente fra di loro, potrebbero esserci infatti nodi intermediari a cui sono stati affidati compiti e servizi differenti, riuscendo a garantire così maggior scalabilità.
L'ultimo principio, ma di fondamentale importanza, prevede che debba esistere un'interfaccia comune di comunicazione fra client e server, permettendo così la semplificazione dell'architettura e che le risorse vengano trasferite in forma standard.

Capitolo 2

BACK-END

In questo capitolo verrà spiegato il funzionamento del framework Spring e di alcuni suoi componenti utilizzati per lo sviluppo del back-end, oltre a una breve descrizione di alcuni strumenti ausiliari.

2.1 Java SE

Java è un linguaggio di programmazione ad alto livello, orientato agli oggetti e progettato per essere il più possibile indipendente dalla piattaforma hardware di esecuzione, grazie alla compilazione in *bytecode* e all'interpretazione poi da parte di una JVM (Java Virtual Machine). Oltre ai concetti generali dei linguaggi orientati agli oggetti, come le classi, le interfacce, il ciclo di vita degli oggetti, l'ereditarietà e il polimorfismo, le collezioni e la gestione delle eccezioni, particolare importanza riveste il ruolo delle *annotations*.

Le *annotations* sono utili sia per la leggibilità del codice da parte degli sviluppatori, ma soprattutto per il compilatore e la JVM perché permettono di esprimere in forma dichiarativa alcune caratteristiche del comportamento che il programma dovrà assumere a tempo di esecuzione.

Figura 2.1: logo Java

Ad oggi, Java risulta essere uno dei linguaggi più popolari ed usati soprattutto per applicazioni *client-server* e proprio il framework Spring si basa su Java SE (Standard Edition).

2.2 Spring

Figura 2.2: logo Spring

Spring è un *framework open-source* che implementa un modello di programmazione e configurazione per le moderne applicazioni *enterprise* basate su Java.
Nacque nel 2003 con lo scopo di semplificare il modello basato su Java EE (Enterprise Edition) e tuttora si conferma essere un *framework* molto leggero grazie alla sua struttura modulare, in quanto è possibile configurarlo a piacimento includendo le componenti necessarie allo sviluppo del progetto e mantenendo anche una facile integrazione con *framework* esterni.
Il nucleo principale di Spring si basa su due concetti chiave:

- *Inversion of Control (IoC)*: è un principio architetturale che inverte il flusso di controllo rispetto alla programmazione procedurale, dove lo sviluppatore si occupa delle operazioni di creazione, inizializzazione ed invocazione dei metodi degli oggetti.
  Grazie all'inversione del flusso di controllo, tutte queste operazioni vengono gestite direttamente dal *framework* stesso tramite l'*IoC Container* che si occupa di istanziare gli oggetti (*beans*) dichiarati nel progetto e di reperire e iniettare tutte le dipendenze ad essi associate.
  Questo principio ha l'obiettivo di rendere le componenti *software* il più indipendenti possibile, in modo da poterle modificare senza che i cambiamenti si riflettano sul resto del sistema, evitando così possibili malfunzionamenti.
- *Dependecy Injection (DI)*: è un processo tramite il quale gli oggetti definiscono e accettano le loro dipendenze attraverso gli argomenti del costruttore o di metodi *setter*. Non sono quindi gli oggetti stessi a creare le proprie dipendenze, ma esse vengono "iniettate" dall'esterno, attraverso l'*IoC Container*.

2.2.1 Spring Boot

Spring Boot è una soluzione per ridurre la complessità di creazione e configurazione dei nuovi progetti basati su Spring, mettendo a disposizione una serie di pacchetti che includono tutte le dipendenze e le impostazioni base per l'avvio delle applicazioni.
Se necessario è possibile modificare le configurazioni di *default*, adattandole alle necessità e ai requisiti del progetto da sviluppare.
La funzionalità con maggior rilievo riguarda la possibilità di includere, all'interno dell'eseguibile, una versione *embedded* di un *web server* (Tomcat, Jetty…) rendendo così l'applicazione indipendente e pronta per l'esecuzione.

2.2.2 Spring Data JPA

Spring Data è uno dei componenti più indispensabili di Spring che implementa un modello standard per l'accesso al livello dei dati e per la gestione della loro persistenza.
Si tratta di un progetto "a ombrello", cioè contenente al suo interno altri sotto-progetti specifici per i vari tipi di database.
Nello specifico, Spring Data JPA permette la comunicazione con *database* relazionali e ne gestisce i dati tramite oggetti Java.
Tramite l'uso di *annotations* è possibile configurare una classe con i suoi campi come un'entità con i suoi attributi, specificandone le possibili associazioni e cardinalità.
Inoltre, rende disponibili metodi personalizzati per l'accesso, l'inserimento, la modifica e la cancellazione dei dati nelle tabelle, senza il bisogno di dichiarare *query* esplicite.

La comunicazione fra livello applicativo e la gestione dei dati sul database avviene tramite un'interfaccia intermediaria, ad esempio Hibernate.
Hibernate è una piattaforma *open-source* che fornisce un servizio di ORM (Object Relational Mapping), ovvero gestisce la persistenza dei dati attraverso la rappresentazione e il mantenimento, su *database* relazionale, di un sistema di oggetti Java.

Il salvataggio dei dati in tabelle è reso possibile da un sistema di gestione di *database* (DBMS) che si occupa anche del mantenimento della loro integrità e della sicurezza.
Un esempio è PostgreSQL, *scelto anche per il progetto finale*, a cui vengono riconosciuti numerosi punti di forza: l'integrità dei dati, l'affidabilità, l'estensibilità, la semplicità di gestione di casi reali, l'utilizzo del linguaggio SQL e infine la *community*, dato che è un progetto *open-source*.

2.2.3 Spring Data REST

Spring Data REST è un componente del framework Spring che permette l'implementazione di servizi REST sulle applicazioni web.

In pratica, rende accessibili le risorse che rappresentano il modello sviluppato, come i dati salvati, e le espone tramite metodi personalizzati e associati ai metodi HTTP.

Supporta inoltre la paginazione tramite link di navigazione, passando anche eventuali parametri negli URL (Uniform Resource Locator) per effettuare richieste più specifiche e filtrare i risultati visualizzabili.

Tutto questo è reso possibile grazie all'utilizzo di specifiche *annotations*, sul codice Java, che personalizzano e configurano i metodi per l'accesso alle risorse esposte da Spring Data REST.

2.3 Strumenti ausiliari

In questa sezione verranno descritti tre strumenti ausiliari, integrati con il *framework* Spring, per lo sviluppo e il test del *back-end*.

2.3.1 Maven

Figura 2.3: logo Maven

Maven è uno strumento *open-source* di gestione di progetti *software* basati su Java e sul concetto di "automazione dello sviluppo", cioè sulla possibilità di rendere automatiche alcune operazioni svolte dagli sviluppatori quotidianamente.

Nello specifico, è caratterizzato dalla presenza di un *file*, in formato XML, denominato POM (Project Object Model) che descrive le dipendenze fra il progetto e le varie librerie esterne necessarie, che vengono scaricate automaticamente in locale, dai vari *repository* remoti.

Questo permette di recuperare in modo uniforme i *file* JAR e di poter spostare il progetto indipendentemente da un ambiente all'altro, avendo la certezza di utilizzare sempre le stesse versioni delle librerie.

2.3.2 Postman

Figura 2.4: logo Postman

Postman è uno strumento per testare le interfacce REST delle applicazioni web.
Permette di effettuare delle richieste a un *server* e di visualizzare le risposte ricevute, senza il bisogno di creare classi di test all'interno del codice del progetto, tramite una comoda interfaccia grafica.
È possibile scegliere il metodo HTTP della chiamata, impostare l'URL di destinazione, configurare gli *header* e il *body* della richiesta e infine di verificare la risposta ottenuta, avendo anche la possibilità di scegliere il tipo di formato di rappresentazione dei dati.

2.3.3 Git

Git è un *software open-source* per il controllo di versione distribuito, realizzato per gestire progetti, anche di grandi dimensioni, in modo efficiente e veloce.
Inizialmente nacque con l'unico obiettivo di tenere traccia, con il passare del tempo, delle modifiche apportate a un progetto da parte dei diversi sviluppatori che, grazie al fatto che era un sistema distribuito, potevano lavorare offline e solo successivamente sincronizzare la loro copia con la copia del *server*.
Questa caratteristica è rimasta ed è diventata uno dei motivi principali per la scelta di utilizzare Git, oltre a molte altre funzionalità.

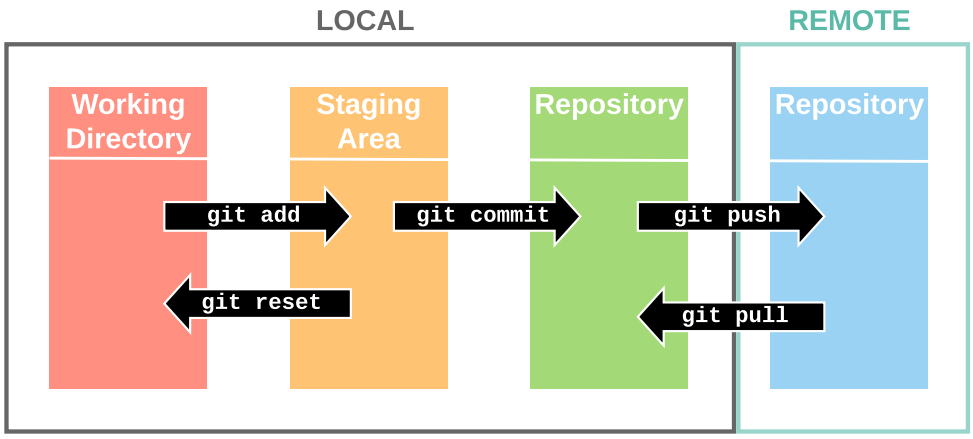

Figura 2.5: schema funzionamento Git

In breve, ogni volta che uno sviluppatore effettua delle modifiche, queste rimangono solo nella cartella di lavoro locale fino a quando non vengono aggiunte alla *staging area*, che è una fase intermedia in cui è possibile rivedere le modifiche prima di un *commit*.
Il *commit* serve per creare un'istantanea di tutti i *file*, modificati e non, e di salvare il tutto nella copia locale del progetto (*repository* locale). Per poter sincronizzare le modifiche con la copia nel *server* del progetto (*repository* remota) è necessario effettuare un'operazione di *push*.
Git permette inoltre la creazione di più rami di lavoro con lo scopo di isolare le modifiche e unirle solo in un secondo momento.

Capitolo 3

FRONT-END

In questo capitolo verrà spiegato il funzionamento del framework Angular e di alcuni suoi componenti utilizzati per lo sviluppo del front-end, oltre a una breve descrizione di alcuni strumenti ausiliari.

3.1 TypeScript

TypeScript è un linguaggio di programmazione *front-end* che estende la sintassi di JavaScript e permette di essere compilato proprio in JavaScript, per essere interpretato correttamente da tutti i *browser web*.

Gli errori tipicamente più diffusi tra i programmatori JavaScript riguardano la tipizzazione dei dati, cioè vi è un'incompatibilità tra il tipo di valore usato e il tipo di valore atteso. Questo può provocare errori in esecuzione o un comportamento inatteso dell'applicazione, in entrambi i casi sarebbe complesso identificare e risalire all'istruzione errata.

TypeScript nasce con l'obiettivo di risolvere questo problema, introducendo delle annotazioni per i tipi primitivi (*number, boolean e string*) e un controllo sui tipi a tempo di compilazione.

Figura 3.1: schema funzionamento TypeScript

Grazie a questa sua funzionalità aggiuntiva, viene considerato come un linguaggio "robusto" e sicuro e quindi adatto allo sviluppo di grandi applicazioni.

Il *framework* Angular usa, come linguaggio di programmazione, TypeScript.

3.2 Angular

Figura 3.2: logo Angular

Angular è un *framework open-source* per creare applicazioni web *front-end* grazie a una serie di funzionalità e strumenti forniti dallo stesso.
È l'evoluzione di AngularJS che utilizzava JavaScript come linguaggio di programmazione, Angular invece si basa su TypeScript.
Offre funzionalità moderne come il supporto nativo a eventi *touch* per dispositivi mobili e la compatibilità con librerie e moduli esterni, garantendo inoltre un apprendimento molto più rapido rispetto al predecessore.
L'architettura modulare consente di strutturare al meglio un'applicazione e di avere un elevato riutilizzo del codice. Permette inoltre un'elevata manutenibilità in quanto ogni componente è adibito ad un'unica funzionalità.

L'architettura di un progetto Angular è costituita da un insieme di moduli che raggruppano funzionalità legate tra loro. Ogni modulo è costituito da un componente a cui è associato un template. Gli elementi principali sono:

- *componenti*: hanno il compito di controllare una determinata porzione di schermo, raccogliere dati e gestire le interazioni da parte dell'utente riguardanti tale area.
- *template*: definiscono la vista di un componente tramite codice HTML (HyperText Markup Language). Il passaggio di dati fra componente e template è chiamato *data binding.*
- *direttive*: sono richiamate dai template e permettono la personalizzazione del codice HTML
- *servizi*: sono insiemi di funzionalità necessarie per il funzionamento dell'applicazione, come il reperimento dei dati dal *server*, lo scambio di informazioni fra componenti, la

gestione degli errori. Mentre i componenti si occupano solamente della raccolta dei dati, i servizi ne definiscono e implementano la gestione.

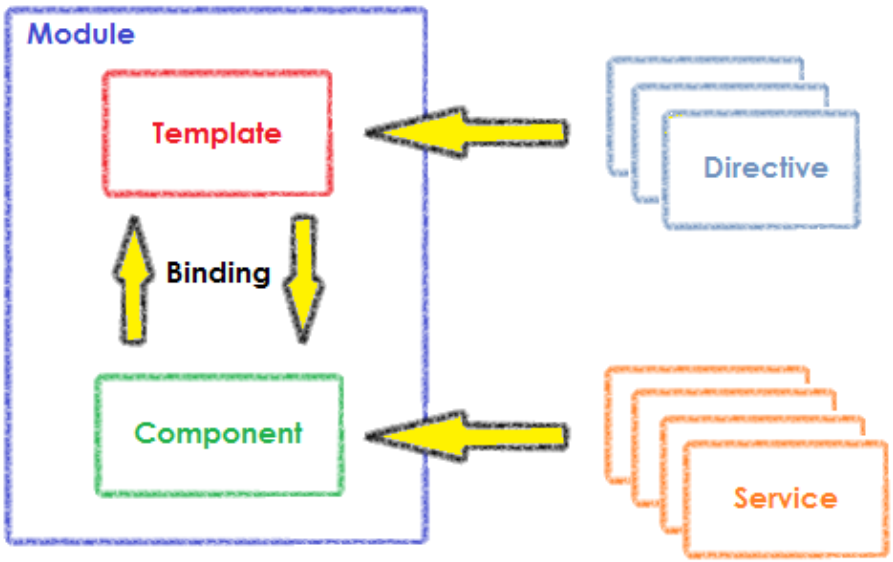

Figura 3.3: schema architettura Angular

È possibile creare una gerarchia di componenti per organizzare al meglio sia la grafica che la logica di una porzione dell'applicazione. Ogni componente sarà associato a un template e potrà richiamare i servizi necessari.
In questo modo si ottiene una maggior riusabilità del codice e una struttura più ordinata, con conseguente maggior facilità nella manutenzione.

3.2.1 Single Page Application

Le applicazioni web sviluppate tramite Angular sono definite *Single Page Application* perché vengono eseguite su una singola pagina del *browser* e il contenuto varia a seconda dell'URL.
In questo modo il *routing* viene effettuato solamente sostituendo i componenti visibili in un determinato momento e i percorsi di navigazione sono dichiarati in un *file* di configurazione dell'applicazione.

3.2.2 Design Pattern

Un design pattern è una soluzione procedurale generale a un problema ricorrente e permette un approccio efficace allo sviluppo di progetti *software* complessi.
Angular supporta alcuni design pattern particolarmente utili per lo sviluppo di applicazioni web e i più importanti sono:

- *Observer*: permette di rimanere in 'ascolto' di determinati eventi. Utilizzato soprattutto con le chiamate ai metodi HTTP che possono richiedere un tempo variabile per ottenere una risposta.
- *Lazy Loading*: permette di ritardare il caricamento di determinati moduli solo in caso di utilizzo. Questo aiuta a mantenere bassa la quantità di dati scaricata dall'utente e a rendere i tempi di caricamento più veloci.
- *Dependecy Injection*: prevede che in ogni componente vengano dichiarati nel costruttore i servizi di cui ha bisogno e in fase di inizializzazione tali dipendenze gli vengono fornite dall'esterno.

3.2.3 Angular CLI

Angular CLI (Command Line Interface) è un'interfaccia a linea di comando per inizializzare, sviluppare, analizzare e mantenere le applicazioni Angular.
Tramite l'utilizzo di comandi appositi è possibile gestire ogni singolo componente dell'applicazione come *node module*, in stile JavaScript.

3.3 Bootstrap

Bootstrap è una libreria di componenti *front-end* per lo sviluppo di siti web *responsive*, cioè che il *layout* delle pagine web si regola dinamicamente, tenendo conto delle caratteristiche del dispositivo utilizzato.
Utilizza costrutti HTML arricchiti da specifiche classi CSS (Cascading Style Sheets) e JavaScript per gestirne il comportamento e per fornire all'utente componenti complessi già formattati come *form*, barre di navigazione, album, *card*…
La versione più recente in forma stabile è la 5.1 che risulta essere molto più "leggera", rispetto alle precedenti, dal punto di vista del codice.

3.4 Node.js

Node.js è un ambiente di esecuzione *open-source* orientato agli eventi per l'esecuzione di codice JavaScript lato *server*.

Tipicamente tale linguaggio di programmazione viene utilizzato lato *client*, in quanto necessita di un *browser* per essere eseguito, ma Node.js permette agli sviluppatori di utilizzarlo anche per generare contenuto dinamico prima ancora che la pagina sia inviata al *browser*.

Ci sono stati dei risvolti positivi anche per il *front-end*, in quanto Node.js ha messo a disposizione degli strumenti, installabili da terminale, per facilitare lo sviluppo di questi progetti.

Un dei moduli più diffusi è sicuramente Npm.

3.5 Npm

Npm (Node Package Manager) è un gestore *open-source* di pacchetti JavaScript che, tramite un'interfaccia a riga di comando (CLI), permette l'installazione di pacchetti, la gestione delle versioni e delle dipendenze.

È composto da un registro, un grande *database* pubblico di applicazioni JavaScript e da un sito web che consente di scoprire i vari pacchetti, pubblici e privati, disponibili.

Il *framework* Angular, Angular CLI e i vari componenti collegati sono gestiti come pacchetti *npm* e distribuiti tramite il registro *npm*.

Capitolo 4

PROGETTO FINALE: NFT STORE

In questo capitolo verrà descritto il progetto finale del tirocinio: un NFT store.

4.1 Descrizione progetto

Il progetto conclusivo del tirocinio consisteva nella progettazione e realizzazione di un'applicazione web tramite il *framework* Spring, per il *back-end*, e Angular, per il *front-end*. Quest'applicazione doveva rappresentare un *NFT store* dove un utente, previa creazione di un account, poteva visualizzare, inserire o vendere i propri *nft* (Non-fungible token).
Altre funzionalità secondarie prevedevano la creazione di un nuovo *account*, la modifica della *password* e la rimozione completa del proprio profilo, *nft* compresi.
È stata adottata una scelta progettuale per la gestione dei *token*, cioè di rappresentare un *nft* come un oggetto legato a un nome e a un'immagine univoca, cercando di emulare, per quanto possibile, il comportamento reale di un gestore di *nft*.

4.2 Rappresentazione dati su database

Una parte importante del progetto riguardava il salvataggio e la gestione dei dati dell'applicazione tramite un *database* relazionale.
Il DBMS scelto è stato PostgreSQL, perfettamente integrabile con Spring tramite il modulo Spring Data JPA.
I dati riguardavano tre entità:

- *utenti*: ogni *account*, associato a un utente, è costituito da un indirizzo *mail*, utilizzato anche come identificatore, e da una *password*.
- *immagini*: identificate tramite un contatore, descritte da un nome, un tipo (formato) e da un *hashcode*. L'attributo *hashcode* permette il salvataggio di una chiave *hash* che rappresenta i dati effettivi dell'immagine; questa scelta è stata effettuata per evitare di immagazzinare quantità eccessive di dati nel *database* e per rendere più efficienti le operazioni di reperimento delle immagini stesse, effettuate dal *back-end*.
- *nft*: identificati tramite un contatore, presentano un nome, un prezzo e un attributo booleano per indicare se è stato venduto oppure è ancora presente. L'entità *nft* è associata a un'immagine, tramite una relazione *uno-a-uno*, e a un utente, tramite una relazione *molti-a-uno*.

4.3 Funzionalità sviluppate – Gestione Account

Le funzionalità dell'applicazione possono essere suddivise in due categorie a seconda che siano legate alla gestione dell'*account* o alle operazioni con gli *nft*.
In questo paragrafo verranno spiegate nel dettaglio le funzionalità legate alla gestione dell'*account*.

Lato *front-end* sono stati implementati due servizi: UserService e AuthService.
UserService fornisce i metodi per la gestione degli *account*, tramite chiamate HTTP, per comunicare con il *back-end* dell'applicazione.
AuthService permette di mantenere salvata la sessione corrente, cioè nel momento in cui un utente effettua il *login* può accedere a funzionalità riservate e perciò è necessario che le richieste successive mantengano salvato l'identificatore dell'*account* attuale.

Lato *back-end* è stato sviluppato un *controller*, UserController, e un servizio, UserService.
UserController predispone un'interfaccia pubblica per i metodi HTTP, associandoli a URL definiti.
UserService definisce nel dettaglio i metodi di gestione dell'*account*, implementandone la logica e riflettendola sul *database*, tramite lo UserRepository.

4.3.1 Registrazione

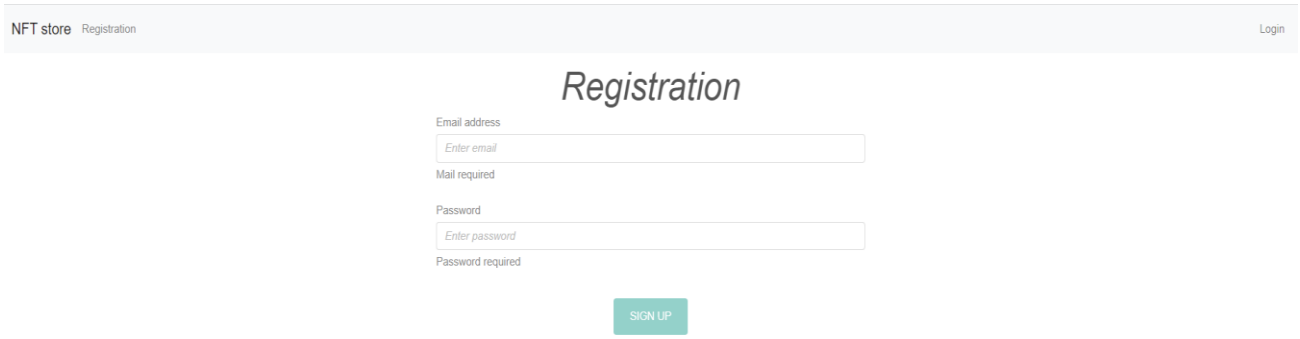

Figura 4.1: maschera di registrazione

La maschera per la registrazione di un nuovo *account* è composta da un *form* che richiede l'inserimento di una *mail* e una *password*. Il componente associato si occupa di invocare il metodo, dello UserService, per l'inserimento e la creazione di un nuovo *account*.

Il *back-end* controlla se la *mail* è associata a un *account* già presente, nel caso invia un messaggio di errore "Bad request", altrimenti restituisce una risposta positiva e inserisce un nuovo *record* nel *database*.
Se la registrazione di un nuovo *account* è andata a buon fine, l'utente viene reindirizzato alla pagina di *login*.

4.3.2 Login

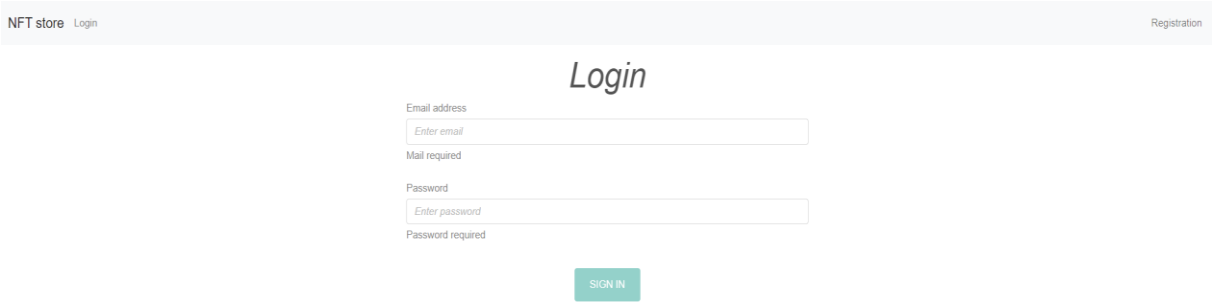

Figura 4.2: maschera di login

La maschera di *login* prevede l'inserimento della *mail* e della *password* associate a un determinato *account*. Il componente associato si occupa di invocare il metodo *login* dello UserService.
Il *back-end* effettua i controlli sull'esistenza di un *account* associato alla *mail* inserita e sulla correttezza della *password*, in caso di errore invia una risposta negativa ("Bad request" o "Not found"), altrimenti restituisce una risposta positiva.
Se il *login* ha avuto esito positivo viene gestita l'autenticazione tramite l'AuhService, che salva nel *local storage* la *mail* dell'account che ha effettuato l'accesso e l'utente viene reindirizzato alla propria *home page* per l'uso effettivo del *NFT store*.
La scelta più ottimale avrebbe dovuto comportare l'uso dei *token* AWT del modulo Spring Security che garantivano maggiore sicurezza e affidabilità, in quanto gestiti dal *back-end*. Purtroppo per motivi di tempo non è stato possibile approfondire questo modulo e questo sistema di autenticazione.

4.3.3 Logout

La funzionalità di *logout* è gestita solamente lato *front-end* in quanto è sufficiente rimuovere dal *local storage* la *mail* dell'*account* che vuole uscire dalla propria area personale.
Successivamente si viene reindirizzati alla pagina di *login*.

4.3.4 Modifica password

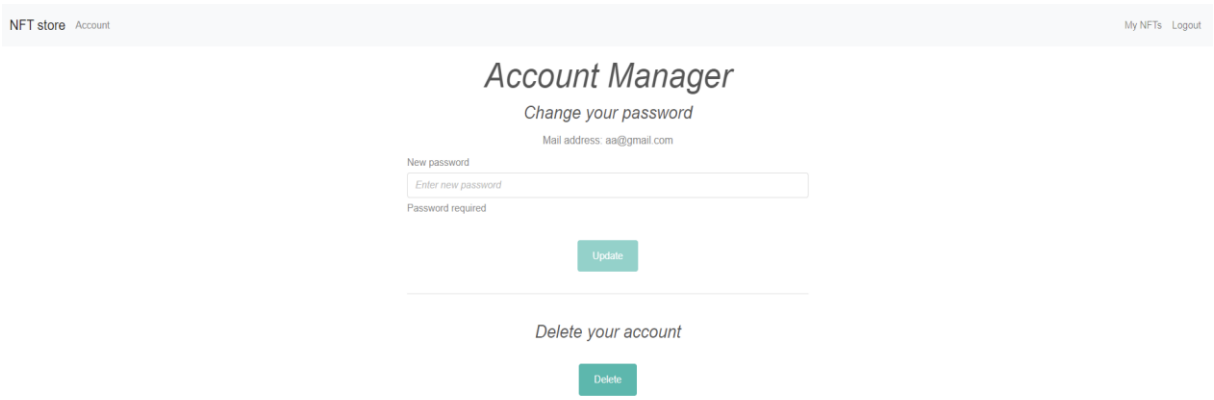

Figura 4.3: maschera di gestione account

La maschera di gestione *account* prevede la possibilità di modificare la *password* attuale con una differente.
Il controllo sulla diversità fra la nuova *password* inserita e quella precedente è effettuato direttamente lato *front-end* e in caso di errore viene mostrato un *alert* di segnalazione.

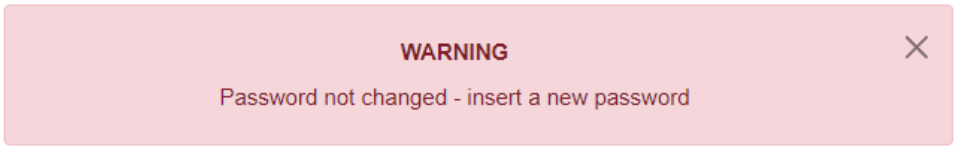

Figura 4.4: alert di errore – password non valida

Se le *password* risultano diverse lo UserService si occupa di effettuare la chiamata al *back-end* che esegue l'operazione effettiva sul *database*.
Se la modifica ha un esito positivo si viene reindirizzati alla pagina di *login* per effettuare l'accesso con i nuovi dati.

4.3.5 Eliminazione account

La maschera di gestione *account* prevede anche la possibilità di eliminare un *account*, compresi gli *nft* associati.

Per effettuare l'operazione, l'utente deve confermare la propria decisione su una finestra apposita, in questo modo lo UserService comunica la scelta al *back-end*.

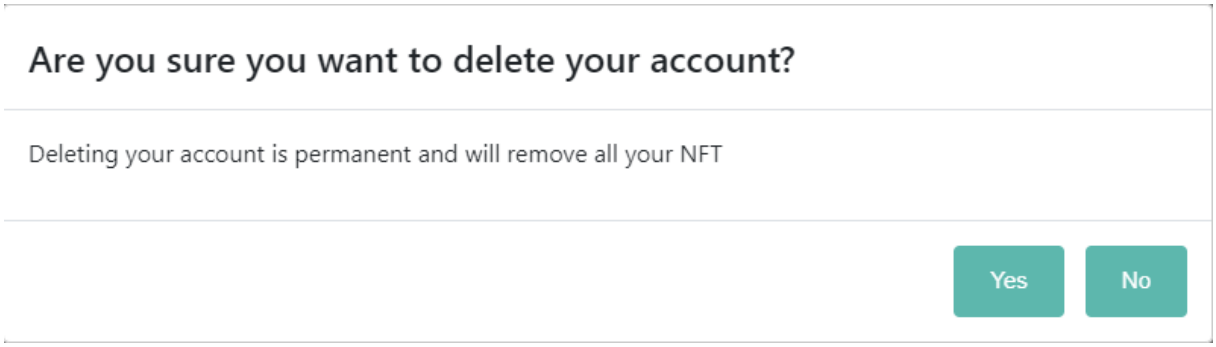

Figura 4.5: conferma eliminazione account

A questo punto, prima di eliminare l'*account* dal *database*, vengono eliminati tutti gli *nft* associati all'utente, immagini comprese.

Al termine l'utente viene reindirizzato alla pagina di *login*.

4.4 Funzionalità sviluppate – Gestione NFT

In questo paragrafo verranno spiegate nel dettaglio le funzionalità legate alla gestione degli *nft*.
Lato *front-end* è stato implementato un servizio: MyNftService
MyNftService fornisce i metodi per la gestione degli *nft*, tramite chiamate HTTP, per comunicare con il *back-end* dell'applicazione.

Lato *back-end* è stato sviluppato un *controller*, MyNftController, e due servizi, MyNftService e ImageService.
MyNftController predispone un'interfaccia pubblica per i metodi HTTP, associandoli a URL definiti.
MyNftService definisce nel dettaglio i metodi di gestione degli *nft*, implementandone la logica e riflettendola sul *database*, tramite MyNftRepository.
ImageService si occupa del caricamento nel *database* delle immagini associate agli *nft*, gestendone la compressione e la generazione di una chiave di *hash*, grazie all'aiuto di ImageRepository.

Le seguenti funzionalità relative alla gestione degli *nft* di un utente sono realizzabili solo dopo aver effettuato l'accesso al proprio *account*.
La *home page* di un *account* mostra gli *nft* propri e permette di aggiungerne altri oppure di venderne.

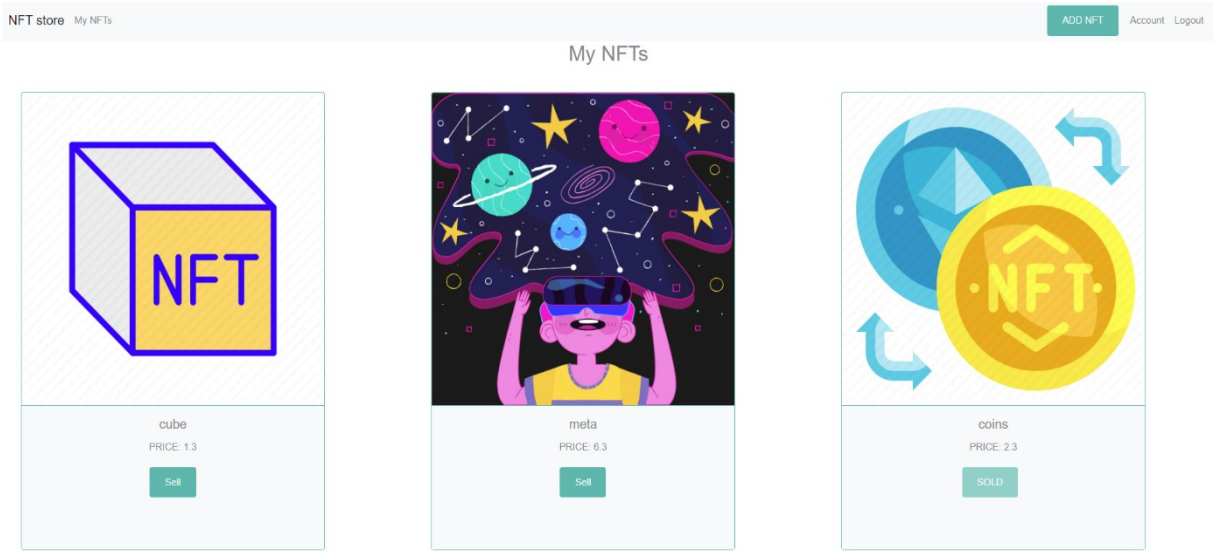

Figura 4.6: home page di un account

4.4.1 Visualizzazione NFT

Quando un utente accede al proprio *account* ha la possibilità di vedere immediatamente la lista dei propri *nft*, disposti tramite una griglia di *card*.

Ogni *card* mostra l'immagine associata al *nft*, il nome, il prezzo e un bottone per abilitare la vendita dell'oggetto.

Il *front-end* si occupa di disporre le *card*, rappresentanti gli *nft*, secondo l'ordine corretto e visualizzandone l'immagine all'interno.

Il *back-end* ha il compito di eseguire un'operazione di *select* per gli *nft* associati a un utente e di decomprimere le immagini prima di inviarle al *browser*.

4.4.2 Aggiunta NFT

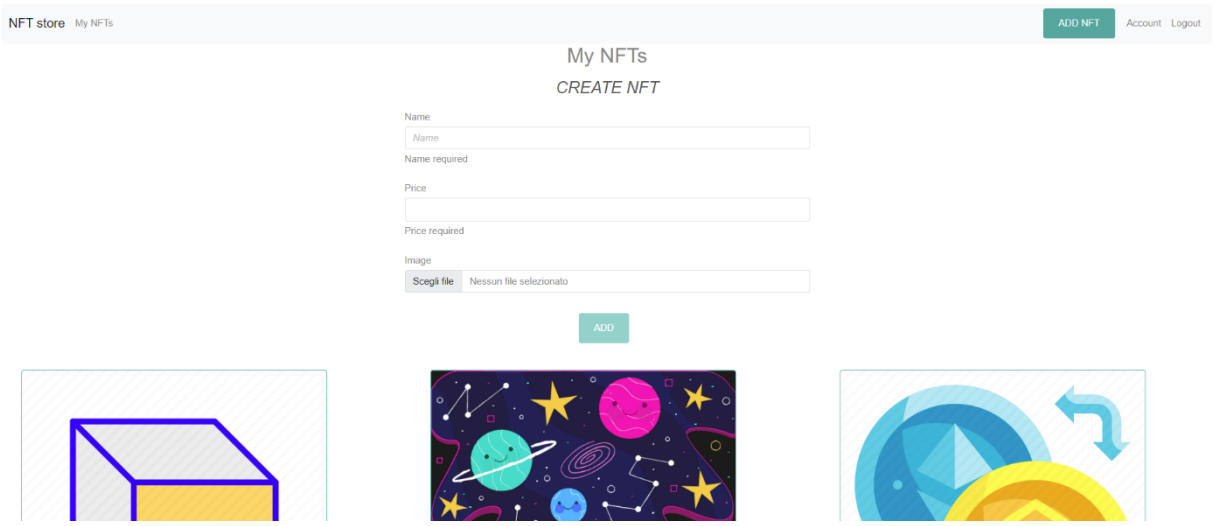

Figura 4.7: form di aggiunta nft

Per l'aggiunta di un *nft* è necessario effettuare un *click* sul bottone posto all'interno della barra di navigazione, in questo modo si verifica l'apertura di un *form* che richiede le caratteristiche del *nft*: il nome, il prezzo e il caricamento di un'immagine.

La prima fase consiste nell'effettuare un controllo, lato *front-end*, sul nome del nuovo *nft* da creare in quanto ogni utente non può avere più *nft* con lo stesso nome. Se viene riscontrato questo problema, viene segnalato all'utente di scegliere un nome differente.

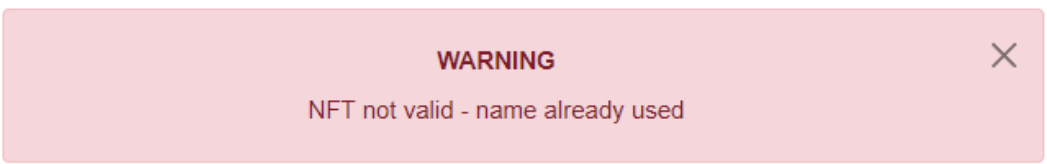

Figura 4.8: alert di errore – nome non valido

Successivamente bisogna salvare l'immagine nel *database* tramite ImageService, che verifica che non esistano altre immagini uguali già caricate, altrimenti restituisce un messaggio di errore.

Figura 4.9: alert di errore – immagine non valida

Se invece il controllo non segnala problemi, l'immagine viene compressa e successivamente generata una chiave *hash* che verrà poi salvata nel *database*.
Al termine di queste operazioni viene chiuso il *form* di inserimento e aggiornata la griglia contenente gli *nft* associati all'utente.

4.4.3 Vendita NFT

La funzione di vendita di un *nft* consiste nell'impostare l'attributo booleano "isSold" a valore *true*. Dopo essere stato venduto, un *nft* rimane comunque visibile, ma risulta disabilitato cioè non è più possibile eseguire nessuna operazione con esso.
La scelta iniziale era di rimuovere semplicemente un *nft*, ma per rendere l'applicazione il più possibile vicina a una situazione reale si è scelto di "emulare" la funzione di vendita, senza però integrare transazioni monetarie perché avrebbero richiesto conoscenze molto avanzate e non in linea con il percorso di tirocinio scelto.
Per fare questo, un utente premendo sul bottone "sell" associato a un *nft* e confermando sulla finestra di validazione, vede lo spostamento dell'oggetto disabilitato all'ultima posizione.

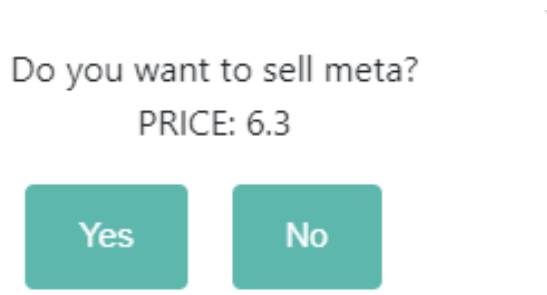

Figura 4.10: conferma vendita nft

MyNftService si occupa di richiamare il metodo lato *back-end* per eseguire l'*update* dell'attributo "isSold" sul *database* e infine mostra nuovamente la lista di *nft* aggiornata.

Capitolo 5
CONCLUSIONI

In questo capitolo sono riportate le considerazioni conclusioni relative all'attività di tirocinio.

L'esperienza di tirocinio svolto presso Sync Lab S.r.l è stata, nel complesso, molto utile, sia dal punto di vista del mio percorso di studi che per quanto riguarda l'accrescimento di nuove competenze professionali.
Le prime settimane di formazione sono state indispensabili per riuscire ad apprendere le conoscenze necessarie alla progettazione e implementazione di un'applicazione web *full-stack*.
L'argomento che ha catturato maggiormente la mia attenzione è stato lo sviluppo del lato *back-end* tramite il *framework* Spring e i suoi numerosi moduli di espansione per la gestione dei *database* e per lo scambio di dati tramite metodi HTTP.
Il progetto *NFT store* ha necessitato di molto lavoro per comprendere al meglio le tecnologie e gli strumenti di sviluppo, per me nuovi, ma ha ottenuto il gradimento del tutor aziendale, avendo tutti i requisiti richiesti e definiti in precedenza.

La necessità di dover apprendere e studiare in autonomia la maggior parte degli argomenti, mi ha mostrato una situazione differente dal contesto universitario e molto più vicina a una realtà aziendale.
Credo che, oltre a quanto evidenziato in precedenza, il valore aggiunto del tirocinio sia proprio la possibilità di conoscere un contesto differente a quello a cui siamo abituati e a mettersi in gioco per crescere sia a livello personale che professionale.

BIBLIOGRAFIA

Siti web consultati

- Scrum: https://www.scrum.org/resources/blog/le-basi-di-scrum
- Principi SOLID: https://it.wikipedia.org/wiki/SOLID
- Servizi REST: https://www.redhat.com/it/topics/api/what-is-a-rest-api
- Spring: https://www.baeldung.com/category/spring/
- Maven: https://maven.apache.org/what-is-maven.html
- Git: https://git-scm.com/about
- TypeScript: https://www.typescriptlang.org/docs/handbook/intro.html
- Angular: https://angular.io/start
- Bootstrap: https://www.bootstrapdash.com/use-bootstrap-4-with-angular/
- Node.js: https://en.wikipedia.org/wiki/Node.js
- Npm: https://www.geekandjob.com/wiki/npm