

# **Documento tecnico Sviluppo maschere front end tramite framework Vue.js**

Margherita Mitillo

## **Sommario**

Questo documento offre una spiegazione tecnica del progetto NFTLab, in particolare tratta lo sviluppo delle maschere front end eseguito con il framework Vue.js

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Il progetto . . . . .	2
<b>2</b>	<b>Progettazione e codifica</b>	<b>3</b>
2.1	Tecnologie utilizzate . . . . .	3
2.1.1	Vue . . . . .	3
2.1.2	Vuetify . . . . .	4
2.1.3	Vuex . . . . .	5
2.2	Organizzazione dei file . . . . .	5
2.3	Chiamate al back end . . . . .	7
2.3.1	CurrentUser.js . . . . .	7
2.3.2	NftService.js . . . . .	9
2.4	Maschere implementate . . . . .	11
2.4.1	Barra di navigazione . . . . .	11
2.4.2	Home . . . . .	11
2.4.3	Visualizza dettagli . . . . .	12
2.4.4	Login . . . . .	12
2.4.5	Registrazione . . . . .	13
2.4.6	Pagina dell'utente . . . . .	14
2.4.7	Upload opera . . . . .	14
2.4.8	Gestione opere . . . . .	15
2.4.9	Modifica dati personali . . . . .	16
2.4.10	Modifica password . . . . .	16

# 1 Introduzione

## 1.1 Il progetto

Il progetto da sviluppare è una web application in ambito Blockchain e NFT. In particolare il sito sviluppato si può definire come simile ad un e-commerce in quanto lo scopo finale è effettuare azioni di compravendita tra i vari utenti iscritti ad esso.

All'interno della web application l'utente può navigare nella pagina principale e visualizzare le opere multimediali poste in vendita, eseguire l'accesso se possiede un account nel sito oppure registrarsi come nuovo account. Dopo essere acceduto al sito, l'utente può modificare i propri dati, inserire una nuova opera multimediale o modificare i dati di quelle precedentemente inserite e comprare un'opera multimediale caricata nel sito da un altro utente.

Essendo un progetto legato al concetto di blockchain e NFT le azioni di compravendita verranno effettuate tramite lo scambio di una moneta virtuale detta Ethereum. Un utente viene identificato tramite l'indirizzo del wallet con cui acquisterà o caricherà una nuova opera multimediale.

Quando un utente carica un'opera multimediale, il file viene salvato tramite codice hash nella blockchain ed in questo modo le opere caricate saranno univoche. Infatti salvando queste informazioni nella blockchain viene creato un timestamp che funge da certificato di attribuzione dell'opera all'utente che carica o compra l'opera multimediale. In questo modo nessuno nessun altro utente può registrare la stessa opera a suo nome poichè una funzione hash produce sempre e solo un risultato rendendo uniche le opere multimediali.

## 2 Progettazione e codifica

### 2.1 Tecnologie utilizzate

#### 2.1.1 Vue



Figura 2.1: Logo di Vue.js

Vue.js è un framework javascript open source nato nel 2013 che presenta un'architettura adottabile in modo incrementale che si concentra sulla composizione dei componenti, inoltre sono presenti funzionalità avanzate offerte tramite librerie e pacchetti di supporto. I componenti Vue estendono gli elementi HTML di base per incapsulare del codice riutilizzabile quindi a livello generale i componenti sono elementi personalizzati a cui il compilatore Vue associa una particolare funzionalità.

Vue utilizza quindi una sintassi basata su HTML e consente di associare il DOM renderizzato ai dati dell'istanza di Vue sottostante. In questo modo i modelli Vue possono essere analizzati da browser e parser HTML conformi alle modifiche ed inoltre con il sistema di reattività Vue è in grado di calcolare il numero minimo di componenti per eseguire nuovamente il rendering applicando la quantità minima di manipolazioni DOM quando cambia lo stato dell'app. Vue presenta un sistema reattivo grazie all'utilizzo di oggetti semplici Javascript ed ad un re-rendering ottimizzato: ogni componente durante il render tiene traccia delle sue dipendenze in modo tale che il sistema sappia quando e di quali componenti deve effettuare nuovamente il render.

Un problema che affligge le web application a pagina singola è che quest'ultime forniscono agli utenti la risposta basata solamente sull'URL dal server, di conseguenza l'utilizzo dei segnalibri a determinate schermate e la condivisione dei collegamenti a sezioni specifiche risulta molto difficile se non impossibile. Vue.js per riuscire a risolvere questo problema fornisce un'interfaccia, detta router, che dà la possibilità di modificare ciò che viene visualizzato sulla pagina in base all'URL indipendentemente da come esso viene modificato. Infatti Vue.js viene fornito con il pacchetto open source "vue-router" che fornisce un'API per aggiornare l'URL dell'applicazione, supportare la cronologia di navigazione e le reimpostazioni di email e password. Tramite questa tipologia di router i componenti devono essere mappati alla route a cui appartengono per indicare dove deve essere eseguito il loro render.

Questo framework implementa il pattern MVVM, acronimo per Model-View-View-Model, una declinazione del più famoso MVC, ovvero Model-View-Controller. I componenti del MVVM sono:

- Model (o Modello): l'implementazione del dominio dati come per il classico Modello del pattern MVC;

- View (o Vista): il componente grafico renderizzato dall'utente formato da HTML e CSS;
- ViewModel (o Vista per il Modello): il collante tra gli altri due componenti, esso fornisce alla View i dati in formato consono alla rappresentazione ed il comportamento di alcuni elementi dinamici.

La grossa differenza tra il pattern implementato da Vue.js e il Model-View-Controller sta nella differenza tra Controller e ViewModel. Il primo, infatti, è una porzione di codice che gestisce la logica di business grazie al Model e ritorna una View da mostrare all'utente; il secondo, invece, rappresenta una versione parallela al Model che risulta essere legato alla View e descrive il comportamento di quest'ultima con funzioni associate. Quindi mentre il Controller esegue logiche di business prima del rendering della View, il ViewModel definisce il comportamento dell'applicazione a runtime.

### 2.1.2 Vuetify

Vuetify è un framework UI completo costruito su Vue.js nel 2014 ed il suo obiettivo è fornire agli sviluppatori gli strumenti per poter creare esperienze utente ricche e coinvolgenti. A differenza di altri framework Vuetify è progettato da zero in modo tale da renderlo facile da imparare ed essere gratificante da padroneggiare con centinaia di componenti realizzate dalle specifiche di Material Design.

Un pregio di questo framework è che adotta un approccio al mobile, questo significa che la web application sviluppata tramite esso sarà pienamente utilizzabile immediatamente su un tablet, un telefono ed un computer. Inoltre è un framework in sviluppo attivo che viene aggiornato settimanalmente rispondendo ai problemi e relativi report della community. Un altro pregio è, come si può vedere nell'immagine sottostante, il gran numero di funzionalità che possiede Vuetify in confronto agli altri framework di Vue.






Vue Framework Comparison 2021					
Features	 Vuetify	 BootstrapVue	 Buefy	 Element UI	 Quasar
Accessibility and section 508 support	●	●	●		
Business and enterprise support	●				
Long-term Support	●				
Release cadence**	Weekly	Bi-Weekly	Bi-Monthly	Bi-Weekly	Bi-Weekly
RTL support	●	●		●	●
Premium themes	●	●			
Treeshaking	Automatic	Manual	Manual	Manual	Automatic
**Based on average of all Major/Minor/Patch releases over the last 12 months.					

Figura 2.2: Funzionalità di Vuetify

### 2.1.3 Vuex

Vuex è una libreria per applicazioni sviluppate in Vue.js e serve come store centralizzato per tutti i componenti dell'applicazione. Esso contiene stati e regole che assicurano che lo stato può essere modificato solo in modo prevedibile.

Inoltre Vuex è uno state management pattern che aiuta a salvare i dati in maniera persistente per poterli utilizzare all'interno dell'applicazione da diversi componenti in maniera efficiente.

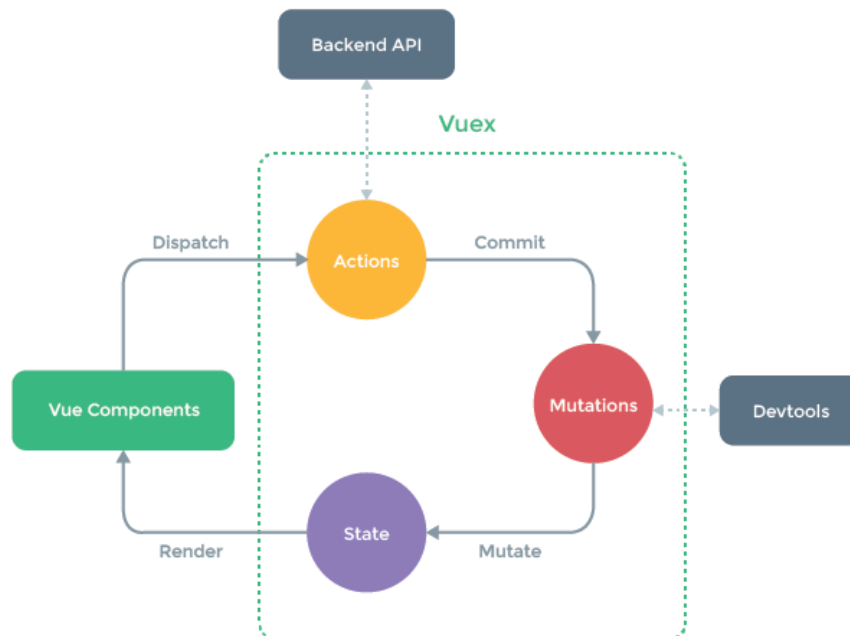


Figura 2.3: Pattern di Vuex

## 2.2 Organizzazione dei file

Quando si crea un progetto per la prima volta tramite il framework Vue.js vengono create di default delle cartelle e dei file per facilitare la progettazione del software. La suddivisione delle cartelle è rappresentata qui di seguito.

```

1 | -- front-end-vue
  | -- codice
  |   |-- .editorconfig
  |   |-- .eslintignore
  |   |-- .eslintrc.js
6  |   |-- .gitignore
  |   |-- .prettierrc.json
  |   |-- .prettierignore
  |   |-- babel.config.js
  |   |-- jest.config.js
11 |   |-- jsconfig.json
  |   |-- package-lock.json

```

```

16 | | -- package.json
| | -- README.md
| | -- vue.config.js
21 | | -- .vscode
| | -- coverage
| | -- dist
| | -- public
| | -- src
| | | -- App.vue
| | | -- main.js
| | | -- style.css
| | | -- assets
26 | | | -- components
| | | -- plugins
| | | -- router
| | | -- store
| | | | -- index.js
31 | | | | -- modules
| | | -- view
| | -- tests
| | | -- unit
| -- docs
| -- documento tecnico
36 | -- directoryList.md

```

La cartella principale si chiama **front-end-vue** ed essa contiene i file di configurazione del progetto, il file **readme**, il file di **gitignore** ed il file **directoryList.md**, creato automaticamente attraverso il comando **mddir** dove viene scritta la struttura delle cartelle del progetto. Inoltre sono presenti tre cartelle:

- **codice**, contenente il codice del progetto;
- **docs**, contenente file YAML utilizzato per la configurazione di Stoplight;
- **documento tecnico**, contenente i file latex utilizzati per la stesura di questo documento.

Per quanto riguarda la cartella legata al codice, essendo la più importante, necessita di una spiegazione degli elementi che la contengono. I file e le sottocartelle create automaticamente alla creazione del progetto però non verranno spiegate, mentre i restanti elementi sono i seguenti:

- **coverage**: cartella che contiene i file e le cartelle create automaticamente quando vengono fatti partire i test di unità;
- **src**: cartella in cui sono presenti file e cartelle necessari per il funzionamento del progetto e sono:
  - il file **App.vue** è la radice dell'applicazione definita nel formato Vue Component e di solito serve a definire il modello della pagina;
  - il file **Main.js** di tipo javascript che inizializza il file **App.vue** ed è responsabile dell'impostazione dei plugin e dei componenti di terze parti da utilizzare per lo sviluppo dell'applicazione;
  - la cartella **assets** contenente le immagini utilizzate all'interno del sito, come quella del logo;
  - la cartella **components** contenente la barra di navigazione (navbar) e le varie finestre d'avviso;
  - la cartella **plugins** contenente i file javascript dei plugin utilizzati per lo sviluppo, in questo caso di Vuetify;
  - la cartella **router** contenente un omonimo file javascript dove vengono definite le route ovvero le varie pagine in cui può navigare l'utente;

- la cartella **store** contenente i file per la configurazione delle chiamate al database e lo state management;
- la cartella **view** contenente le possibili schermate che può visualizzare l'utente;
- la cartella **tests** che contiene la sottocartella **unit** dove sono presenti i test di unità effettuati.

*Nelle sezioni di seguito verranno spiegate in dettaglio il funzionamento delle chiamate al back end e delle varie maschere implementate con le loro criticità*

## 2.3 Chiamate al back end

Il file principale, **index.js**, contiene la configurazione di Vuex e del persisted state con l'inclusione dei moduli che sono di numero tanti quanti sono i servizi del back end che vanno ad invocare. Per rendere più chiaro ed efficiente il lavoro ho creato quindi il file **CurrentUser.js**, legato al servizio del back end dell'utente, e il file **NftService.js**, legato al servizio del back end degli NFT.

Entrambi i file hanno la struttura uguale e quello che cambia è il contenuto, infatti sono caratterizzati da quattro elementi costanti:

- state, contenente le risorse che verranno utilizzate nei metodi;
- actions, contenente i metodi dove vengono fatte le invocazioni ai servizi del backend tramite axios;
- mutations, contenente i metodi che modificano le risorse, sono come un equivalente dei metodi set;
- getters, contenente i metodi che ritornano i valori presenti nelle risorse.

Gli state del file **CurrentUser.js** li ho resi persistenti, in questo modo potranno essere utilizzati all'interno della web application, per esempio dopo che l'utente ha eseguito l'autenticazione utilizzo i suoi dati per visualizzare il suo nome nel bottone che lo porta alla pagina personale. Queste informazioni possono essere utilizzate per fare dei controlli che decidono se reindirizzare o meno certe parti della web application.

Essendo legati a servizi diversi i metodi e le chiamate al back end dei due file sono molto diversi e nelle sezioni successive verranno spiegate nel dettaglio.

### 2.3.1 CurrentUser.js

In questo file ho implementato diversi servizi legati all'utente. I servizi in questione sono:

- **autenticazione:** l'utente prova ad effettuare il login e nella componente legata all'autenticazione avviene una chiamata al metodo **LoginUser** passando come parametro l'utente.

```

1  loginUser({ commit }, user) {
2    axios
3      .post(urlBackend + 'login', {
4        email: user.email,
5        password: user.password
6      })
7      .then(response => {
8        commit('setUser', response.data);
9        commit('setLoggedIn');
10       localStorage.setItem('user', JSON.stringify(response.data));
11       router.push('/');
12     })
13     .catch(error => {
14       commit('setErrorMessageLog', error.response.status);
15     });
16   }

```



Come si può vedere dall'immagine soprastante viene eseguita una chiamata POST al database specificando i dati dell'utente che verranno inviati. Se la chiamata va a buon fine i dati dell'utente autenticato verranno salvati sia nello stato persistente che nel localStorage del web e si verrà reindirizzati alla homepage. Se invece la chiamata non va a buon fine viene effettuato il catch dell'errore e viene settato un messaggio di errore che può variare in base all'errore che ritorna il back end.

- **registrazione**

```

4      signUp({ commit }, user) {
        axios
          .post(urlBackend + 'signup', {
            email: user.email,
            password: user.password,
            name: user.name,
            surname: user.surname,
            dob: user.dob,
            wallet: user.wallet
          })
          .then(response => {
            commit('setUser', response.data);
            commit('setLoggedIn');
            localStorage.setItem('user', JSON.stringify(response.data));
            router.push('/');
          })
          .catch(error => {
            commit('setMessageErrorSig', error.response.status);
          });
19    }

```

Come si può vedere dall'immagine soprastante viene eseguita anche per la registrazione una chiamata POST al database specificando i dati dell'utente che verranno inviati, di numero maggiore rispetto ai dati per l'autenticazione. Se la chiamata va a buon fine i dati dell'utente autenticato verranno salvati sia nello stato persistente che nel localStorage del web e si verrà reindirizzati alla homepage. Se invece la chiamata non va a buon fine viene effettuato il catch dell'errore e viene settato un messaggio di errore che può variare in base all'errore che ritorna il back end.

- **logout**

```

      logOut({ commit }) {
        commit('setLoggedOut');
        localStorage.clear('user');
      }

```

Come si può vedere dall'immagine soprastante questo è l'unico metodo che non possiede chiamate effettive al back end, infatti quando l'utente vuole effettuare il log out verrà semplicemente chiamato un metodo che fa ritornare l'utente alla homepage e cancella i dati dell'utente nello stato persistente. Infine viene svuotato il localStorage dai dati dell'utente.

- **modifica dei dati personali**

```

1      updateUser({ commit }, user) {
        var url = urlStop + 'user/${JSON.parse(localStorage.getItem('user')).id}';
        console.log(user);
        axios
          .put(url, {
            password: user.password,
            name: user.name,
            surname: user.surname,

```

```

11     email: user.email,
        dob: user.dob,
        wallet: user.wallet
    })
    .then(response => {
        commit('setUser', response.data);
        localStorage.setItem('user', JSON.stringify(response.data));
16    }).catch(error =>{
        commit('setErrorMessageMod', error.response.status);
    })
}
```

Come si può vedere dall'immagine soprastante viene eseguita la modifica dei dati personali tramite una chiamata PUT al database specificando i dati dell'utente che verranno inviati e inviando come parametro l'id identificativo dell'utente. Se la chiamata va a buon fine i dati aggiornati verranno salvati sia nello stato persistente che nel localStorage del web. Se invece la chiamata non va a buon fine viene effettuato il catch dell'errore e viene settato un messaggio di errore che può variare in base all'errore che ritorna dal back end.

- **modifica della password**

```

1    updatePassword({}, user) {
        axios.put(urlStop + 'user/password', {
            email: user.email,
            oldPassword: user.oldPassword,
            newPassword: user.newPassword
6        }).catch(error =>{
            commit('setErrorMessageMod', error.response.status);
        })
    }
}
```

Come si può vedere dall'immagine soprastante viene eseguita la modifica della password tramite una chiamata PUT al database. Essendo un dato che non deve essere utilizzato all'interno del sito per questioni di sicurezza, il database non ritorna alcuna risposta. Se invece la chiamata non va a buon fine viene effettuato il catch dell'errore e viene settato un messaggio di errore che può variare in base all'errore che ritorna dal back end.

## 2.3.2 NftService.js

In questo file ho implementato diversi servizi legati agli NFT. I servizi in questione sono:

- **opere dell'utente utente**

```

1    userOperas({ commit }) {
        var url =
            urlBackEnd + 'nft/user/${JSON.parse(localStorage.getItem('user')).id}';
        axios.get(url).then(response => {
            commit('setOperas', response.data);
6        });
    }
}
```

Come si può vedere dall'immagine soprastante questo metodo serve per ritornare le opere dell'utente. Questo avviene tramite una chiamata GET al database fornendo come parametro l'id identificativo dell'utente.

- **categorie disponibili**

```

3   getCategories({ commit }) {
      axios.get(urlBackEnd + 'categories'+examples).then(response => {
        commit('setCategories', response.data);
      });
    }

```

Come si può vedere dall'immagine soprastante questo metodo serve per ritornare le categorie disponibili nel database tramite una chiamata GET al database. Questo metodo viene invocato ogni qual volta serve sapere le categorie presenti, quindi ad esempio per caricare una nuova opera, per modificarne una o per la creazione del filtro presente nella homepage.

#### • caricamento di una nuova opera

```

5   uploadOpera({ commit }, opera) {
      var url =
      urlBackEnd + 'nft/user/${JSON.parse(localStorage.getItem('user')).id}';
      axios
      .post(url, opera)
      .then(response => {
        commit('setOpera', response.data);
        router.push('/');
      })
10    .catch(error => {
      commit('setErrorOpera', error.response.data);
    });
  }

```

Come si può vedere dall'immagine soprastante questo metodo serve per caricare una nuova opera nel database. Questo avviene tramite una chiamata POST al database fornendo come dati le informazioni della nuova opera e come parametro l'id identificativo dell'utente. Se invece la chiamata non va a buon fine viene effettuato il catch dell'errore e viene settato un messaggio di errore che può variare in base all'errore che ritorna dal back end.

#### • modifica di un'opera esistente

```

2   updateOpera({ commit }, opera) {
      var url =
      urlStop + 'nft/user/${JSON.parse(localStorage.getItem('user')).id}';
      axios
      .put(url, {
7        id: opera.id,
        description: opera.description,
        title: opera.title,
        price: Number(opera.price),
        currency: 'ETH',
12       type: opera.type,
        author: opera.author,
        owner: opera.owner,
        categories: opera.categories,
        path: opera.path
      })
17    .then(response => {
      commit('setOpera', response.data);
    })
    .catch(error => {
22       commit('setErrorOpera', error.response.data);
    });
  }

```

Come si può vedere dall'immagine soprastante questo metodo serve per modificare un'opera già presente nel database. Questo avviene tramite una chiamata PUT al database fornendo come dati le nuove informazioni dell'opera e come parametro l'id identificativo dell'utente. Se invece la chiamata non va a buon fine viene effettuato il catch dell'errore e viene settato un messaggio di errore che può variare in base all'errore che ritorna dal back end.

- home

```
2  getHomeOperas({ commit }) {
    axios.get(urlBackend + 'nft'+examples).then(response => {
      commit('setHomeOperas', response.data);
    });
  }
```

Come si può vedere dall'immagine soprastante questo metodo serve per ottenere tutte le opere presenti nel database per mostrarle nella homepage. Questo avviene tramite una chiamata GET al database.

## 2.4 Maschere implementate

Per comprendere meglio il funzionamento delle maschere queste verranno spiegate nelle sezioni sottostanti nel dettaglio.

### 2.4.1 Barra di navigazione

Questa componente è presente in quasi tutto il sito ad eccezione delle pagine di Login, di Registrazione e di Upload dell'opera. Per poter implementare questo dettaglio il componente che ha il compito di reindirizzarla, ovvero il file **App.vue**, possiede una condizione dove viene fatto il controllo se la pagina in cui si sta navigando sia una di quelle dove deve essere presente o meno e di conseguenza verrà reindirizzata o meno. Se il controllo va a buon fine la Navbar verrà reindirizzata, altrimenti non verrà mostrata all'utente.

Un'altra particolarità di questo componente è la presenza di due bottoni che variano in funzione se l'utente è autenticato o meno nel sito. Infatti se l'utente non si autentica si vedranno due bottoni per effettuare il login o la registrazione, mentre se esso è autenticato si bottoni serviranno per visualizzare la pagina personale oppure effettuare il logout. Per gestire questa particolarità è stato implementato un controllo condizionale dove controlla appunto lo stato dell'utente tramite una variabile chiamata *isLogged*, che assume il valore a vero se autenticato.

### 2.4.2 Home

Questa è la pagina iniziale tramite la quale l'utente può visualizzare tutte le opere presenti nel database con integrata la paginazione, inoltre è presente una combobox che dà la possibilità di filtrare le opere per categoria. Infine è presente un bottone "Visualizza dettagli" che ha una funzionalità che verrà spiegata in seguito.

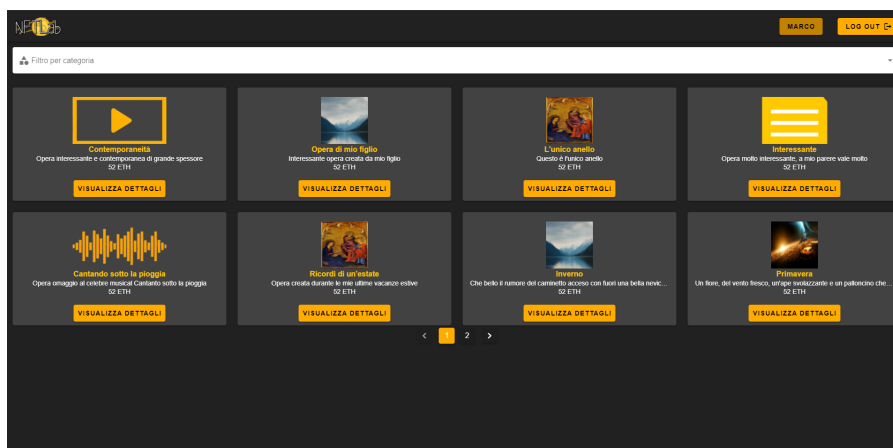


Figura 2.4: Home Page

### 2.4.3 Visualizza dettagli

Questa funzionalità fa apparire una finestra di dialogo a schermo intero che mostra tutte le informazioni dettagliate dell'opera selezionata. Nella parte sinistra l'utente potrà vedere una preview dell'opera, mentre nella parte destra l'utente potrà vedere: titolo, prezzo con valuta, descrizione, autore, proprietario e categorie di appartenenza.

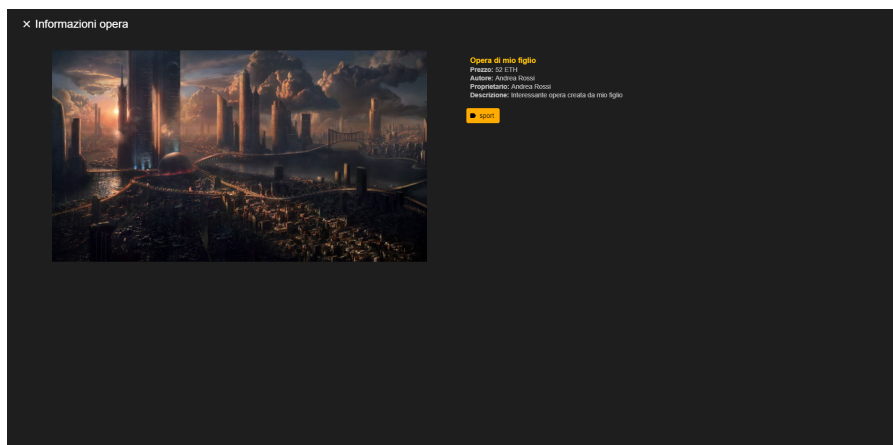


Figura 2.5: Finestra di dialogo delle informazioni dell'opera

Questa finestra di dialogo è stata progettata a schermo intero in modo tale da poter aggiungere ulteriori informazioni o funzionalità senza dover modificare la struttura della finestra di dialogo.

### 2.4.4 Login

In questa pagina l'utente può effettuare l'autenticazione nel sito tramite email e password. In questi due campi sono state inserite delle regole che, se non rispettate, generano un errore visualizzabile tramite una scritta sotto il campo che si sta modificando. Le regole in questione sono:

- entrambi i campi sono obbligatori, quindi devono essere tutti compilati;
- l'email inserita deve essere valida  
*esempio di email valida: nome@email.it;*
- la password deve avere una lunghezza di minimo otto caratteri di cui almeno una lettera maiuscola, una minuscola, un numero e una carattere speciale.

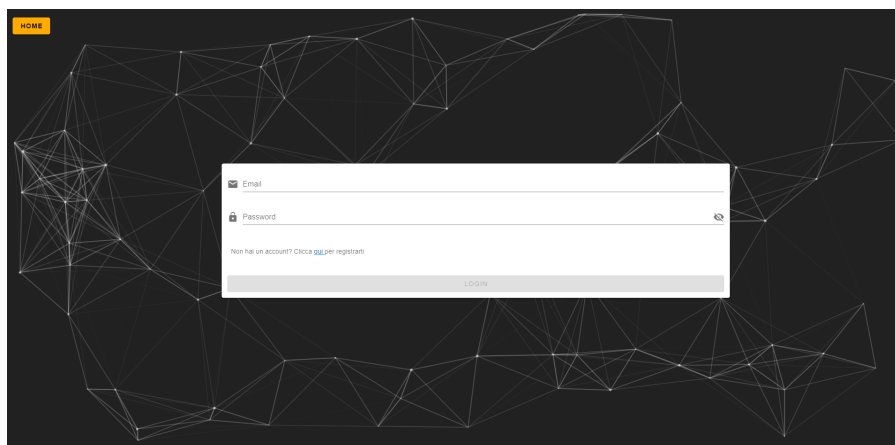


Figura 2.6: Pagina di login

In alto a sinistra è presente un bottone per tornare alla home. Inoltre il bottone per validare l'autenticazione è disabilitato fino a quando l'utente non compila entrambi i campi per evitare chiamate al back end con dati mancanti.

### 2.4.5 Registrazione

In questa pagine l'utente può effettuare la registrazione nel sito inserendo: nome, cognome, email, data di nascita, wallet address, password e conferma password. Similmente alla pagina di login sono state inserite delle regole che, se non rispettate, generano un errore visualizzabile tramite una scritta sotto il campo che si sta modificando. Le regole in questione sono:

- tutti i campi sono obbligatori, quindi devono essere tutti compilati;
- l'email inserita deve essere valida  
*esempio di email valida: nome@email.it;*
- la data di nascita deve corrispondere ad una persona maggiorenne;
- il wallett address inserito deve essere valido  
*esempio di wallett address valido: 0x390c7a1EA64e521B725b1869Ea054DDBF05F9abe;*
- la password deve avere una lunghezza di minimo otto caratteri di cui almeno una lettera maiuscola, una minuscola, un numero e una carattere speciale;
- la conferma della password deve essere identica alla password appena inserita.

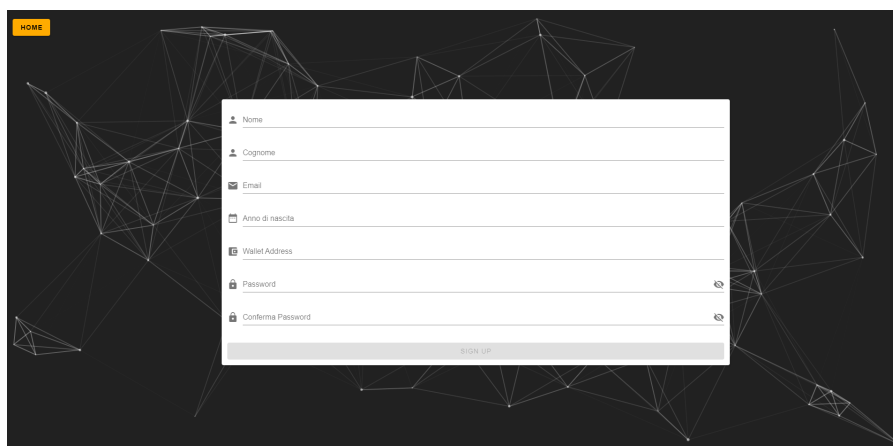


Figura 2.7: Pagina di registrazione

Come per la pagina di autenticazione, in alto a sinistra è presente un bottone per tornare alla home. Inoltre il bottone per validare l'autenticazione è disabilitato fino a quando l'utente non compila entrambi i campi per evitare chiamate al back end con dati mancanti.

## 2.4.6 Pagina dell'utente

In questa pagina l'utente può visualizzare le sue informazioni personali e le opere che possiede. Nel momento che verranno modificate le informazioni da parte dell'utente, queste verranno modificate a schermo automaticamente.

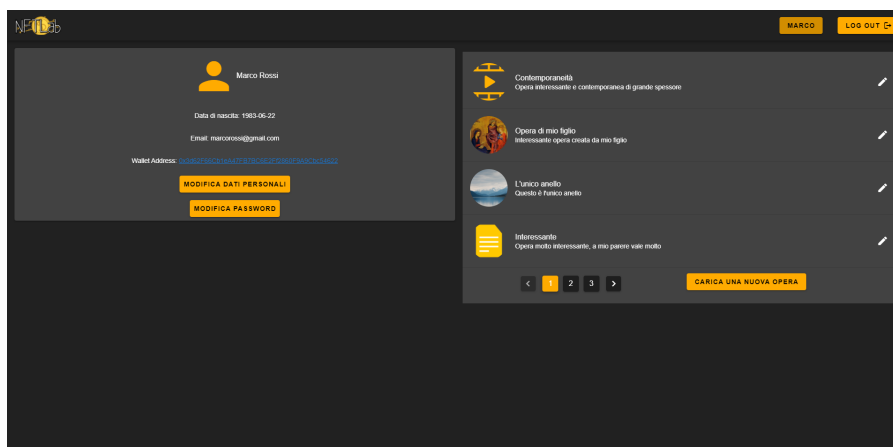


Figura 2.8: Pagina personale dell'utente

## 2.4.7 Upload opera

In questa pagina è possibile per l'utente aggiungere una nuova opera. I campi da compilare sono: titolo, descrizione, file, prezzo. Quando l'utente carica un file verranno generati automaticamente una preview del file specifica in base al tipo del file inserito e un campo non modificabile che indica il tipo del file inserito.

Similmente alla pagina di registrazione sono state inserite delle regole che, se non rispettate, generano un errore visualizzabile tramite una scritta sotto il campo che si sta modificando. Le regole in questione sono:

- tutti i campi sono obbligatori, quindi devono essere tutti compilati.

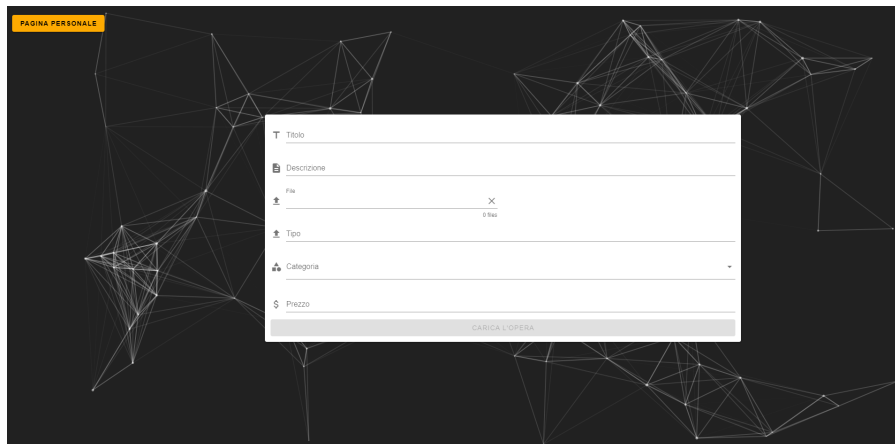


Figura 2.9: Pagina di upload dell'opera

Nel momento in cui l'utente carica il file verrà generata un'immagine in base al tipo di file: se viene caricata un'immagine si potrà visualizzare una *preview* di essa invece se viene caricata un'altra tipologia di file verrà generata un'immagine che indica la tipologia.

## 2.4.8 Gestione opere

L'utente può unicamente modificare una sua opera ma non può cancellarla. Questo è dovuto al fatto che al momento dell'upload l'opera verrà caricata sulla blockchain rendendola un elemento unico ed irripetibile quindi non può cancellarla. Per motivi analoghi l'utente ha delle restrizioni anche nella modifica dell'opera; infatti di essa può modificare solo: titolo, descrizione, prezzo e categorie di appartenenza. Un altro campo non modificabile è il tipo del file caricato.



Figura 2.10: Finestra di dialogo per la modifica dell'opera



### 2.4.9 Modifica dati personali

L'utente, dopo aver premuto un bottone con la dicitura "Modifica dati personali", vedrà apparire una finestra di avviso dove può modificare i propri dati. Per motivi di univocità l'utente non potrà modificare l'email con cui ha fatto l'accesso, infatti potrà modificare solamente il nome ed il cognome. Inoltre per motivi di sicurezza il bottone per confermare la modifica è disabilitato di default e viene data la possibilità di confermare la modifica solo dopo aver inserito la password corrente.

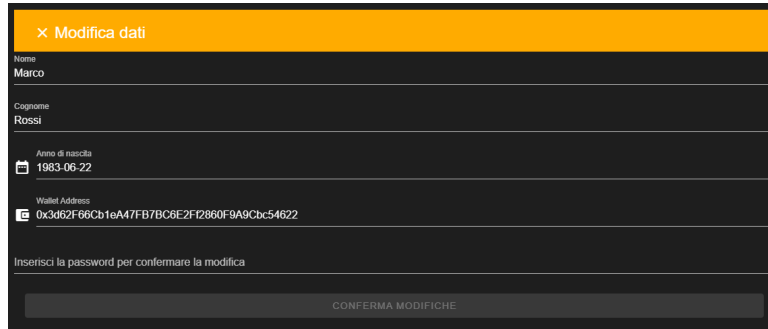


Figura 2.11: Finestra di dialogo per la modifica dei dati

### 2.4.10 Modifica password

L'utente dopo aver premuto un bottone con la dicitura "Modifica password", vedrà apparire una finestra di avviso dove può modificare la password. Per poterlo fare, per motivi di sicurezza, l'utente dovrà inserire la sua email, la password corrente e la nuova password. Similmente alla modifica dei dati personali, il bottone per confermare la modifica rimarrà disattivato fino a quando l'utente non avrà completato tutti i campi.

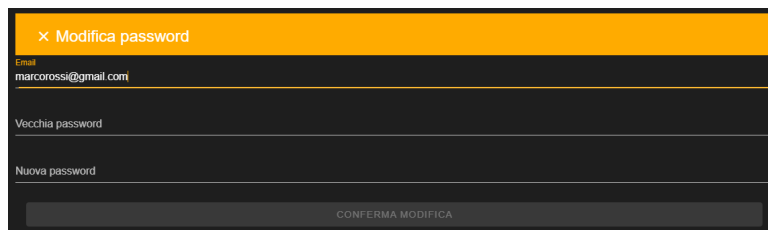


Figura 2.12: Finestra di dialogo per la modifica della password