



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato del corso di:
SOFTWARE ARCHITECTURES AND METHODOLOGIES – 9 CFU
2023-2024

MyPortfolio:

Implementazione di una piattaforma Backend + Frontend
con architettura API RESTful

Andrea Neri – 7060638

Docenti:
Enrico Vicario
Jacopo Parri
Samuele Sampietro

Sommario

Sommario	2
Introduzione	5
Obiettivi e intenti dell'elaborato	5
Analisi dei requisiti	6
Vincoli	6
Requisiti non funzionali	6
Requisiti funzionali	6
Analisi di progettazione	8
Gestione e organizzazione delle immagini	8
Accesso e permessi utente	8
Visualizzazione e Copyright delle Immagini	8
Sezione Shop (estensione del progetto)	9
Considerazioni sulla scalabilità e manutenibilità	9
Domain Model	9
Analisi diagrammi di progetto	10
UML Diagram	11
Deployment Diagram	12
Frontend	13
Backend	13
Sequence Diagram	14
Use Case Diagram	17
Tabella di copertura dei casi d'uso	19
Descrizione dei package	21
Implementazione lato JAVA	22
it.myportfolio.configuration.myportfolioApplication.java	23
Annotations	26
it.myportfolio.model	27
Work	27
ImageProject	28
ShopableImage	29
User	30
ERole	31

Role	32
Cart	32
SalesOrder.....	33
it.myportfolio.dto	34
UserPersonalDetailsDTO vs User	35
SignupRequest	36
DetailsSalesOrderDTO.....	37
SimpleShopableImageDTO.....	38
it.myportfolio.mapper.....	38
it.myportfolio.service	39
it.myportfolio.repository	43
Spring Data JPA	44
JpaRepository	44
Metodi CRUD.....	45
Query Methods	45
ImageRepository	46
WorkRepository	46
UserRepository.....	47
it.myportfolio.controller	47
Gestione Copyright delle immagini.....	49
API security: gestione autenticazione e autorizzazione nelle API	50
WebSecurityConfig	54
UserDetailsServiceImpl	57
UserDetailsImpl.....	57
AuthTokenFilter.....	58
AuthEntryPointJwt.....	59
AuthController.....	60
JwtUtils.....	61
API security: gestione sicurezza immagini.....	62
Descrizione componenti di utility	63
Implementazione lato DB.....	65
Implementazione front-end	68
Struttura del front-end.....	69
AllOrders.vue.....	71
shop.service.ts	72
Index.....	72

Contatti.....	73
Funzionalità User.....	73
Registrazione.....	73
Login.....	74
Gestione profilo.....	74
Visualizzazione Gallerie fotografiche.....	75
Shop.....	77
Funzionalità Admin	81
Inserimento nuovo lavoro	81
Aggiunta immagini ad un Work	81
Gestione Work e immagini contenute	82
Gestione visibilità work	84
Gestione immagine nella sezione shop	85
Storico ordine effettuati	87
Verifica HASH ordini.....	87
Sezione shop e blockchain.....	87
Descrizione piattaforma.....	88
Network e principale e fork utilizzati.....	89
Implementazione e flusso.....	90
Script python.....	91
Libreria Web3 e transaction type.....	94
Componenti lato Back-End	95
API.....	100

Introduzione

L'idea del progetto nasce da una richiesta di un amico fotografo che voleva risolvere la problematica relativa alla condivisione in modo selettivo e con autenticazione alcune cartelle del suo NAS, contenenti i sample di progetti fotografici con collaboratori, clienti e futuri clienti.

Il primo tentativo ipotizzato è stato quello di sfruttare una delle funzionalità del NAS, ovvero l'accesso ai file tramite una app messa a disposizione dal produttore. Inoltre, il NAS permette la creazione di utenze, con la possibilità di definire l'accesso alle share folder in modo granulare. Seppure comoda come soluzione, il dover condividere sulla rete pubblica l'intera library (se pur passando dai server/app fornite dal produttore) del NAS, è stata scartata.

Il secondo tentativo, prima dell'idea di questo progetto ed ancora in uso, era l'invio tramite piattaforme come WeTransfer (e simili) dei sample dei clienti e dei futuri clienti; questa modalità risulta molto scomoda, lenta e temporanea, in quanto il link per il download ha una scadenza.

Le richieste fondamentali erano la possibilità di far visualizzare immagini protette da watermark (copyright) ad utenti registrati ed ai quali erano state fornite le autorizzazioni di visualizzazione.

La catalogazione delle foto sarà gestita tramite raccolte (definite Work) ognuna delle quali rappresenterà un singolo progetto fotografico (es. shooting fotografico per l'azienda ACME corp.) o un insieme di lavoro accumulati da una stessa caratteristica (es. fotografia paesaggistica), questo permetterà una gestione fine degli accessi.

Obiettivi e intenti dell'elaborato

- Fornire un sito web fruibile sia da PC che smartphone, che oltre ad avere una homepage, una sezione di presentazione e una di contatto, avesse un'area dedicata a cui accedere solo tramite login (username e password)
- Fornire una piattaforma per la condivisione selettiva e autenticata di shooting fotografici raggruppati in Work (questo concetto verrà esposto nelle prossime sezioni).
- Avesse una visualizzazione in base al ruolo assegnato all'utente che si autentica
- L'utenza standard, una volta registrato e loggato, potrà accedere in visualizzazione alla/e cartella/e in base alle autorizzazioni fornite dall'amministratore.
- La visualizzazione della foto sarà tramite una galleria fotografica (una per ogni cartella o area di lavoro al quale quell'utente ha accesso) sfruttando le thumbnail in modo da rendere più veloce il caricamento della pagina, che al click verranno mostrate a tutto schermo.

Volendo stilare un elenco delle funzionalità/sezioni minime da sviluppare e rendere disponibili lato front-end per l'utilizzatore finale, queste potrebbero essere riassunte così:

- Homepage,
- Pagina di presentazione
- Pagina contatti
- Login/Registrazione
- Pannello di amministrazione per poter caricare nuove foto e cancellare/modificare quelle già presenti
- Pannello di amministratore per poter gestire i permessi di accesso sulle cartelle

- Sezione galleria in cui verranno mostrate le foto presenti in una singola cartella
- Realizzazione delle gallerie fotografiche tramite thumbnail in modo da rendere il caricamento delle pagine più veloci
- Sia le thumbnail che le immagini a dimensioni reali dovranno presentare un watermark per proteggere gli scatti da eventuali furti.

Come estensione al progetto di base (che rimarrà solo per scopi didattici) è stato pensato di aggiungere una sezione Shop. In questa sezione l'amministratore potrà inserire un certo numero di foto che vorrà vendere (in copia unica, essendo opere artistiche). Gli utenti una volta loggati potranno inserire nel carrello le foto che vorranno e procedere all'acquisto.

L'acquisto verrà registrato all'interno di blockchain per certificare l'acquisto. Quest'ultima caratteristica è stata realizzata tramite Rest API fornire da una libreria esterna Open Source che registra le transazioni su un fork della blockchain Ethereum.

Analisi dei requisiti

Vincoli

- V-1. Le foto sono già presenti in un NAS su rete LAN privata. Le immagini non devono essere caricate sul Database come Blob, ma verranno registrati solo gli URL.
- V-2. Le foto dovranno essere necessariamente caricate in formati fotografici standard (PNG, JPEG, JPG, BMP). Non potranno essere caricati i cosiddetti "scatto grezzo", formati fotografici non renderizzati come NEF, CR2, CR3, ARW e RAF.

Requisiti non funzionali

RNF-1. Architettura del Sistema

- a. L'architettura del sistema deve essere basata sul modello MVC.
- b. Il back-end deve essere sviluppato utilizzando JakartaEE con il framework Spring.
- c. Il front-end deve essere sviluppato utilizzando Vue JS e Bootstrap.
- d. L'architettura deve garantire la portabilità del sistema.

RNF-2. Gestione del Copyright

- a. Il sistema deve garantire il rispetto del copyright sulle opere

Requisiti funzionali

RF-1. Tipologie utenze e azioni permesse

- a. L'applicazione deve prevedere 2 tipologie di utenza: Admin e User
- b. L'utenza con Role User deve avere la possibilità di visualizzare i work a cui gli è stato fornito l'accesso
- c. L'utenza con Role Admin deve aver la possibilità di visualizzare tutti i work disponibili
- d. L'utenza con Role Admin deve aver la possibilità di accedere ai pannelli di amministrazione sia della sezione Galleria che Shop
- e. L'utenza con Role User deve accedere e fare acquisti nella sezione Shop

RF-2. Gestione pannello di controllo dell'amministratore

- a. L'amministratore deve poter accedere a un pannello di controllo.
- b. Dal pannello di controllo, l'amministratore deve poter caricare e gestire risorse (immagini).
- c. Dal pannello di controllo, l'amministratore deve poter gestire i permessi di lettura per i singoli "work".

RF-3. Gestione immagini

- a. Le risorse caricate devono poter essere raggruppate per "work" (es. shooting fotografico per l'azienda ACME corp. o un insieme di lavori accomunati da una stessa caratteristica, come fotografia paesaggistica).
- b. L'applicativo deve supportare il caricamento, la catalogazione e il recupero di risorse immagine nei formati PNG, JPEG, JPG e BMP.
- c. Il sistema deve generare automaticamente le miniature (thumbnail) per ogni immagine caricata per ottimizzare i tempi di caricamento delle pagine.
- d. La creazione delle miniature deve avvenire al click del pulsante "Genera Thumbnail" e le miniature devono essere salvate nella sottodirectory 'thumbnail' nella stessa cartella delle immagini originali.
- e. Il sistema deve garantire l'applicazione di un watermark on demand dal lato back-end, nel momento in cui viene richiesta la visualizzazione di un'immagine a risoluzione standard
- f. Il sistema deve permettere di applicare in modo statico un watermark sulle thumbnail

RF-4. Gestione fine dei permessi di autorizzazione e accesso

- a. Gli utenti devono registrarsi alla piattaforma per accedere alle risorse.
- b. Ogni utente deve avere autorizzazioni mirate su cartelle specifiche.
- c. Alcune immagini utilizzate nella homepage devono avere visibilità pubblica senza necessità di login.
- d. L'accesso alle foto private deve avvenire solo dopo login dell'utente.
- e. L'utente deve poter visualizzare solo le foto per cui ha i diritti di accesso.

RF-5. Componenti Java di ausilio

- a. Il sistema deve includere componenti Java per la generazione automatica delle miniature delle immagini.
- b. Il sistema deve consentire l'aggiunta di un watermark personalizzato alle risorse immagine su richiesta (on demand).

RF-6. Sezione Shop e gestione acquisti

- a. L'applicativo deve prevedere una sezione "Shop" dove saranno presenti alcune immagini selezionate dall'amministratore, simili a NFT.
- b. Gli utenti devono poter acquistare le immagini presenti nella sezione "Shop".
- c. Gli acquisti devono essere registrati su un sistema blockchain utilizzando le API fornite da un provider di tecnologia blockchain.
- d. L'amministratore deve visualizzare tutti gli ordini fatti all'interno dello shop
- e. L'amministratore può caricare e gestire ShopableImage
- f. Gli utenti devono poter visualizzare i propri acquisti

Analisi di progettazione

L'obiettivo principale è la creazione di un sistema efficiente e sicuro per la gestione, visualizzazione e vendita delle immagini, sfruttando le tecnologie moderne e garantendo la protezione dei diritti d'autore.

Gestione e organizzazione delle immagini

Poiché le foto sono già presenti sul NAS, si è deciso di non caricarle su un database come BLOB (Binary Large Object), verranno registrati solo gli URL delle immagini all'interno di un campo testuale nel database. Questo approccio offre diversi vantaggi:

- Efficienza del database: poiché le query dovranno recuperare solo stringhe testuali, si ha una riduzione del carico di lavoro sul database, migliorando le prestazioni complessive del sistema.
- Minor occupazione di spazio: registrando sul database solo gli URL delle foto si è evitato di avere un Database grande in termini di occupazione di memoria
- Accesso rapido: Gli URL consentono un accesso diretto e rapido ai file immagine, migliorando l'esperienza utente.
- Tempistiche di Backup ridotte.

Per garantire la sicurezza e la corretta visualizzazione delle immagini, sarà implementata una ACL (Access Control List) ad hoc sul NAS. Questa lista di controllo degli accessi permetterà di definire l'indirizzo IP sorgente del server dove sarà installato l'applicativo.

Per una gestione più pratica, le singole immagini verranno caricate e organizzate raggruppandole per "work". Un "work" può essere definito come un progetto fotografico (es. shooting fotografico per l'azienda ACME corp.) o un insieme di lavori accomunati da una stessa caratteristica (es. fotografia paesaggistica). Questa organizzazione permetterà una più facile gestione delle risorse e un accesso più intuitivo per gli utenti, ottenendo:

- Facilità di navigazione: gli utenti a trovare facilmente le risorse di cui hanno bisogno.
- Gestione efficace: gli amministratori potranno gestire in modo efficace e organizzato le immagini caricate
- Gestione fine dei permessi di visualizzazione

Accesso e permessi utente

Gli utenti dovranno registrarsi alla piattaforma per accedere alle risorse. Una volta completata la registrazione ed effettuato il login, l'amministratore (utente con ruolo Admin) potrà concedere i permessi di visualizzazione dei "work" agli utenti. Sarà predisposta una sezione contatti all'interno dell'applicazione, dove gli utenti potranno fare richieste di visualizzazione, fornendo una descrizione della tipologia di immagini che desiderano vedere.

Visualizzazione e Copyright delle Immagini

Gli utenti potranno visualizzare solo le immagini per cui hanno i diritti di accesso tramite una gallery.

La pagina verrà carica utilizzando i thumbnail generati staticamente, salvati nella sub-directory e

con il watermark già presente.

Per le immagini a dimensione reale si è deciso di non applicare il watermark in modo statico, ma per garantire il rispetto del copyright, si è deciso di generare le immagini con watermark lato back-end. Questo approccio eviterà di dover salvare sul NAS sia le foto originali che quelle con watermark, riducendo l'occupazione di spazio e semplificando la gestione delle risorse. Le immagini con watermark saranno generate dinamicamente quando richieste, garantendo così la protezione dei diritti d'autore senza compromettere la qualità del servizio.

Sezione Shop (estensione del progetto)

Come estensione del progetto, che non sarà richiesta nell'applicativo che andrà in produzione inizialmente, si realizzerà una sezione shop. In questa sezione, l'amministratore potrà caricare immagini da vendere come opere uniche. Gli utenti potranno acquistare queste immagini, e ogni acquisto sarà registrato su un sistema blockchain (nelle prossime sezioni analizzeremo questo aspetto). Questo processo sfrutterà API messe a disposizione da un provider di tecnologia blockchain, garantendo la tracciabilità e la sicurezza delle transazioni.

Considerazioni sulla scalabilità e manutenibilità

Il sistema è progettato per essere scalabile e manutenibile. Utilizzando un'architettura basata su JakartaEE con il framework Spring per il back-end e Vue JS con Bootstrap per il front-end, si garantisce una separazione chiara delle responsabilità e una facile manutenibilità.

- **Scalabilità orizzontale:** Il sistema può essere scalato orizzontalmente aggiungendo più server man mano che la base di utenti cresce. Questa capacità garantisce che le prestazioni del sistema rimangano ottimali anche con un numero crescente di utenti e carichi di lavoro.
- **Modularità:** La modularità del sistema permette di aggiungere nuove funzionalità senza compromettere quelle esistenti. I componenti del sistema sono progettati per essere indipendenti, facilitando l'integrazione di nuove caratteristiche e miglioramenti in modo sicuro ed efficiente.
- **Aggiornamenti facili:** L'architettura moderna facilita l'aggiornamento del sistema con nuove tecnologie e miglioramenti. Le tecnologie adottate, come Spring e Vue JS, sono ampiamente supportate e aggiornate, garantendo che il sistema possa evolvere e incorporare rapidamente nuove funzionalità e patch di sicurezza.
- **Applicazione Responsiva:** Utilizzando Bootstrap per il front-end, l'applicazione è completamente responsive. Questo significa che l'interfaccia utente si adatta automaticamente a diverse dimensioni di schermo e dispositivi, garantendo un'esperienza utente ottimale sia su desktop che su dispositivi mobili.
- **Creazione di App native:** Utilizzando le API, è possibile creare app native per Android e iPhone senza dover utilizzare un browser.

Domain Model

Dopo avere definito tutte quelle che sono le richieste e i requisiti dell'utilizzatore finale, un ulteriore passo da considerare è l'analisi del *domain model* (modello di dominio), che fornisce a tutti

coloro che devono lavorare su un sistema, una base comune di concetti su cui ragionare e una terminologia condivisa rigorosa e specifica.

In questa analisi si definiscono modelli concettuali (UML, Deployment Diagram, Use Case, Sequence Diagram) che forniranno una guida per la realizzazione del codice.

A partire quindi da vincoli, requisiti non funzionali e requisiti funzionali sono state estrapolate le seguenti entità:

- **User:** per rappresentare un utente all'interno del sistema.
- **Work:** rappresenta un lavoro fotografico o meglio un contenitore logico per raggruppare più immagini legate a uno stesso progetto fotografico. La sua responsabilità principale è gestire i metadati del progetto (committente, titolo, data conclusione) e fornire un contesto per gli ImageProject associati.
- **ImageProject:** rappresenta ogni scatto fotografico all'interno di un progetto specifico (Work). La sua responsabilità è quella di contenere le informazioni relative all'immagine (percorso file, titolo). ImageProject è un'entità dipendente da Work ed è strettamente associata ad esso. Non può esistere senza un Work di riferimento, e il suo ruolo nel dominio è quello di rappresentare la parte concreta e atomica del lavoro fotografico.
- **ShopableImage:** rappresenta una specializzazione di un ImageProject arricchita da informazioni aggiuntive per essere venduta nello shop online; un'immagine disponibile per l'acquisto all'interno dello shop.
- **Cart:** rappresenta il carrello dell'utente all'interno della sezione shop.
- **SalesOrder:** rappresenta un ordine di acquisto.
- **Role:** rappresenta il ruolo di un utente (attualmente i ruoli presenti sono Admin e User).

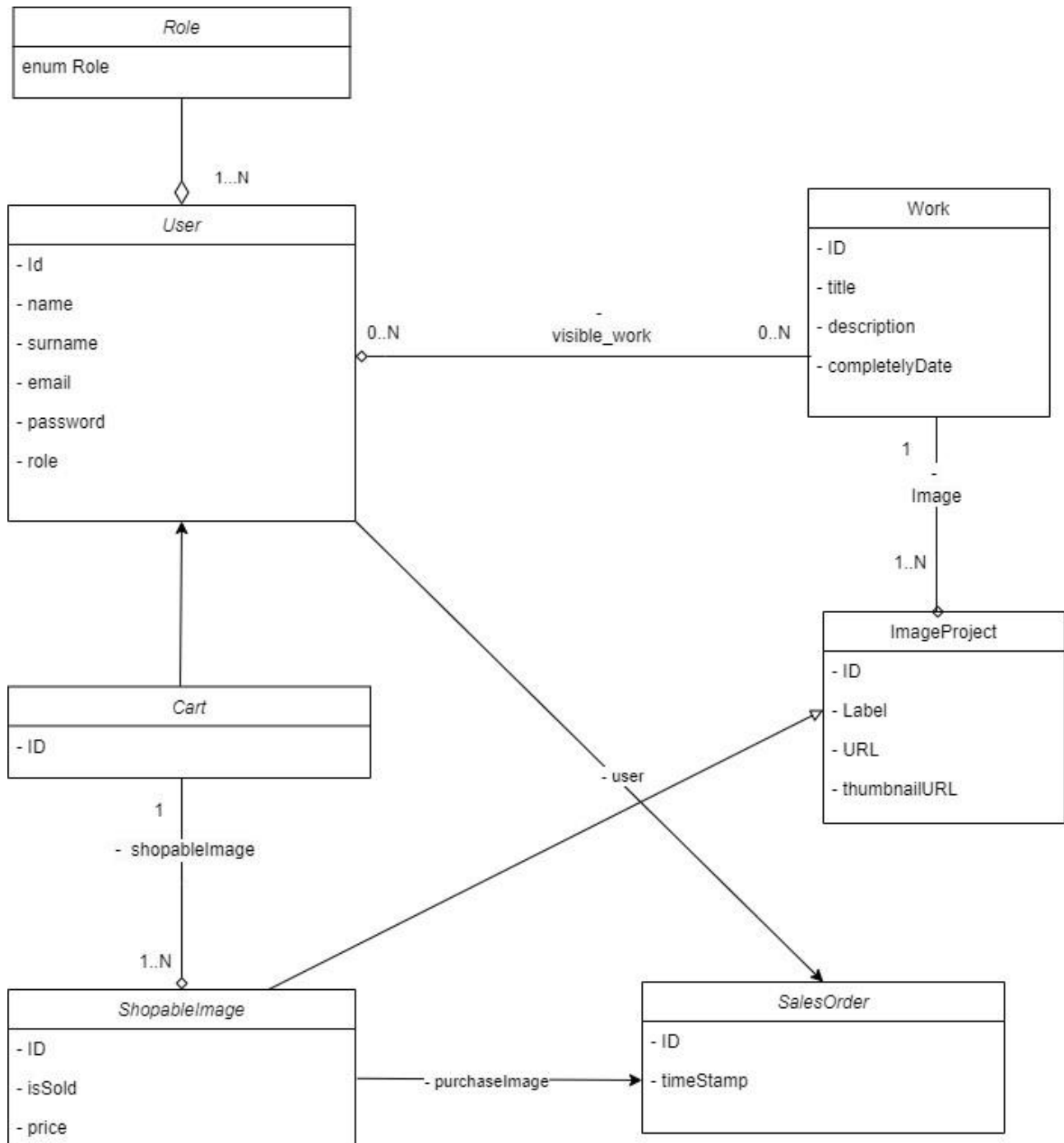
e le seguenti relazioni tra entità:

- Ad un User possono essere associati uno o più Role
- Un User ha una relazione di visibilità con Work (associazione 0..N).
- Ogni Work contiene ImageProject (aggregazione 1..N).
- Ogni User ha un Cart (associazione 1..1).
- Cart contiene ShopableImage (associazione 1..N).
- SalesOrder contiene ShopableImage (associazione 1..N).
- Un User può aver associato un SalesOrder (associazione 0..N).

Analisi diagrammi di progetto

A partire dall'analisi fatta per il domain model sono stati realizzati i 4 diagrammi seguenti.

UML Diagram



Descrizione entità:

- **Role**: rappresenta i ruoli che uno User può avere. Grazie a questo attributo è possibile gestire le autorizzazioni sulle chiamate API in modo selettivo. Attualmente i ruoli configurati sono 2: Admin e User.
- **User**: modello generico dell'utente. Ha una relazione N-N con la classe Role, poiché un utente può avere entrambi i ruoli. Ha una relazione N-N con la classe Work, risolta a livello database dalla tabella intermedia *visibleWork*. Grazie a questo attributo è possibile gestire i

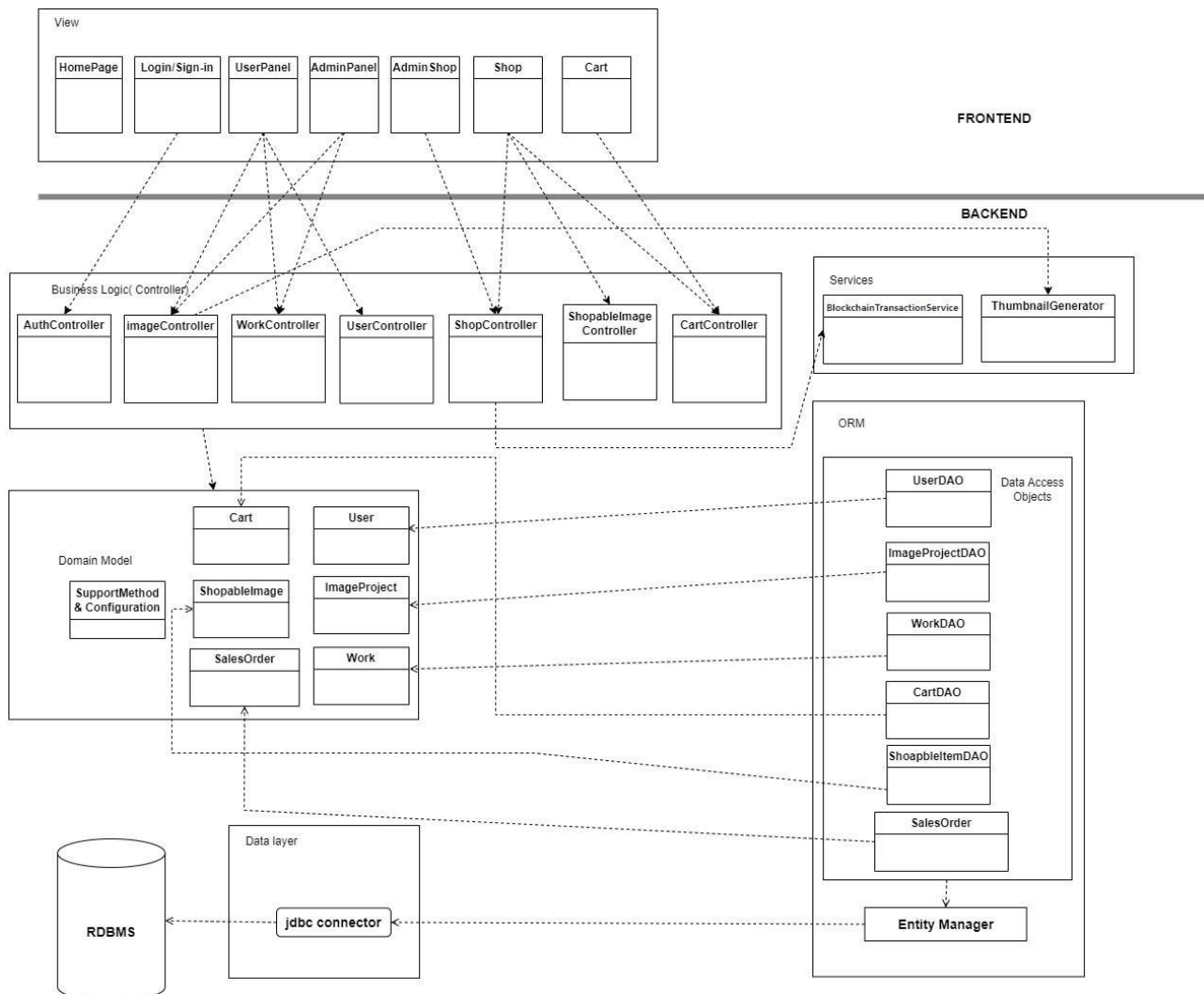
permessi di visualizzazione dei Work.

Ogni utente ha associato un oggetto Cart, che rappresenta il carrello temporaneo (prima di procedere con l'acquisto vero e proprio) nello shop.

- **Work:** modello che permette di raggruppare le ImageProject appartenenti allo stesso shooting o accomunate da una stessa caratteristica
- **ImageProject:** rappresenta i singoli scatti. Tramite gli attributi URL e thumbnailURL sarà possibile "scaricarle dal NAS" e visualizzarle sul Client
- **Cart:** permette la memorizzazione temporanea delle opere che un utente vorrebbe comprare
- **ShoppableImage:** rappresenta le "opere" vendibili. Questa classe è ottenuta come estensione della classe ImageProject. Vengono aggiunti i parametri isSold e price.
- **SalesOrder:** classe che permette la registrazione delle varie vendite, memorizzando chi e quando sono state acquistate determinate opere.

Parte di queste informazioni verranno successivamente salvate in una Blockchain.

Deployment Diagram



Grazie al deployment diagram possiamo vedere i componenti principali dell'architettura del sistema.

Possiamo notare la suddivisione netta tra i componenti di front-end e quelli di back-end, le relazioni tra i vari componenti e come si interfacciano tra di loro.

Frontend

Il front-end è composto da diverse pagine html (viste), che grazie al framework VueJS, interagiscono con i vari controller presenti nel back-end per fornire funzionalità all'utente finale. L'interazione avviene tramite chiamate API RESTful con tre livelli di autenticazione in base al Role dell'utente (nessun ruolo, User, Admin).

- **HomePage:** la pagina principale dell'applicazione, da dove è possibile recuperare le varie informazioni del fotografo ed iniziare l'interazione con l'applicazione.
- **Login/Sign-in:** pagina per l'autenticazione e la registrazione degli utenti.
- **UserPanel:** gli utenti "loggati" da questa pagina potranno visualizzare le gallerie fotografiche per le quali hanno i diritti di visualizzazione.
- **AdminPanel:** pannello di controllo per gli amministratori. Questa è la sezione principale per gli amministratori del sistema: da qui potranno creare/modificare/cancellare le raccolte fotografiche (Work), potranno caricare/modificare/cancellare immagini e potranno fornire o revocare i permessi di visualizzazione agli utenti.
- **Admin Shop:** pannello di controllo per gli amministratori per la gestione dello Shop.
- **Shop:** Sezione per la visualizzazione e l'acquisto di immagini da parte dell'utenza.
- **Cart:** Sezione per la gestione del carrello degli acquisti e la visualizzazione dei propri acquisti.

Backend

Il back-end è composto dai Controller che espongono API, dai servizi (classi di ausilio), dal Domain Model, dall' ORM, dal JDBC per l'interazione con il Database e il DB MySQL per la memorizzazione dei dati.

Business Logic (Controller)

- **AuthController:** gestisce l'autenticazione e la registrazione degli utenti.
- **ImageController:** gestisce le operazioni relative alle immagini.
- **WorkController:** gestisce le operazioni relative ai lavori (work).
- **UserController:** gestisce le operazioni relative agli utenti.
- **ShopController:** gestisce le operazioni relative al negozio online.
- **ShopableImageController:** gestisce le operazioni relative alle immagini acquistabili.
- **CartController:** gestisce le operazioni relative al carrello.

Services

- **BlockchainTransactionService:** Gestisce le transazioni sulla blockchain per la vendita delle immagini.
- **ThumbnailGenerator:** Genera le miniature delle immagini presenti sul NAS. Contiene anche la funzione per l'applicazione del watermark sulle immagini

Nel **Domain Model** sono realizzate le Entities, che permettono di rappresentare la semantica degli oggetti di dominio e i **relativi Data Access Objects (DAO)** per la gestione della persistenza delle entità nel database.

SupportMethod & Configuration contiene i metodi di supporto e configurazioni dell'applicazione

Il livello dei dati o **Data Layer** ha il compito di interfacciarsi con il database relazionale (RDBMS) tramite un connettore JDBC per eseguire operazioni di lettura e scrittura (operazioni CRUD).

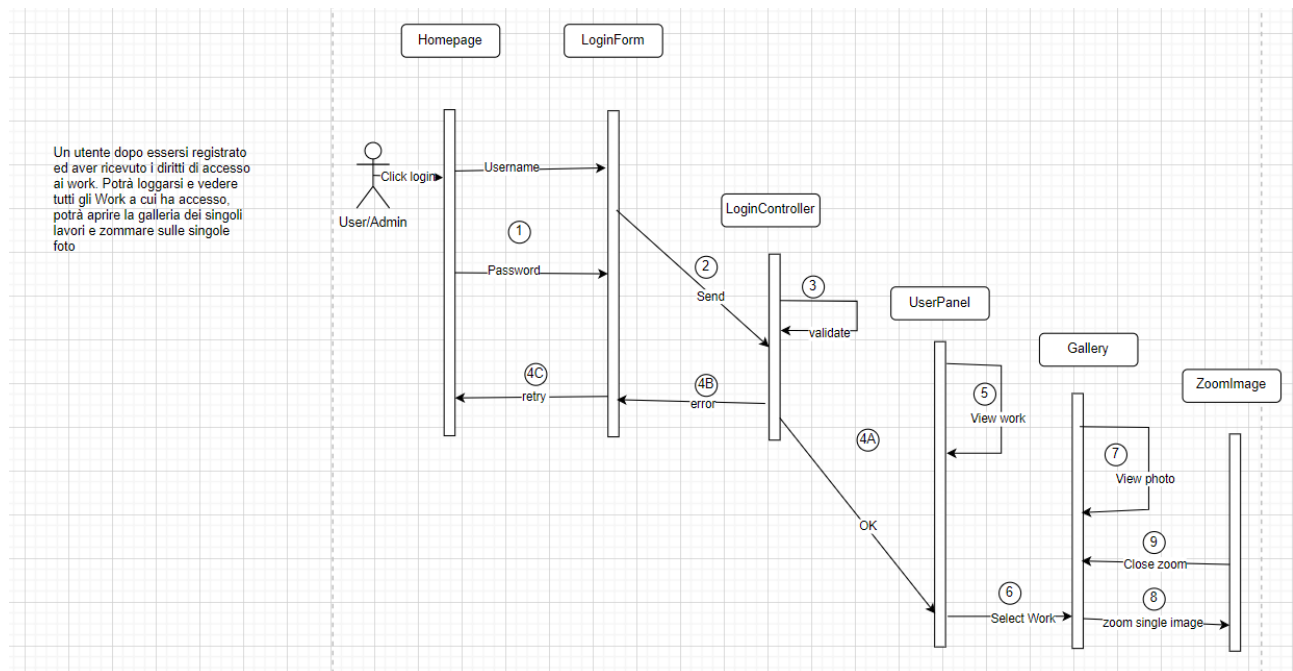
Grazie a questa suddivisione, si ottengono due vantaggi molto importanti:

- **Separazione delle responsabilità:** l'architettura del sistema garantisce una chiara separazione delle responsabilità tra le varie componenti, facilitando la manutenibilità e la scalabilità del sistema.
- **Modularità:** la modularità del sistema permette di aggiungere nuove funzionalità e componenti senza compromettere quelle esistenti, grazie alla suddivisione in controllori, servizi e DAO.

Sequence Diagram

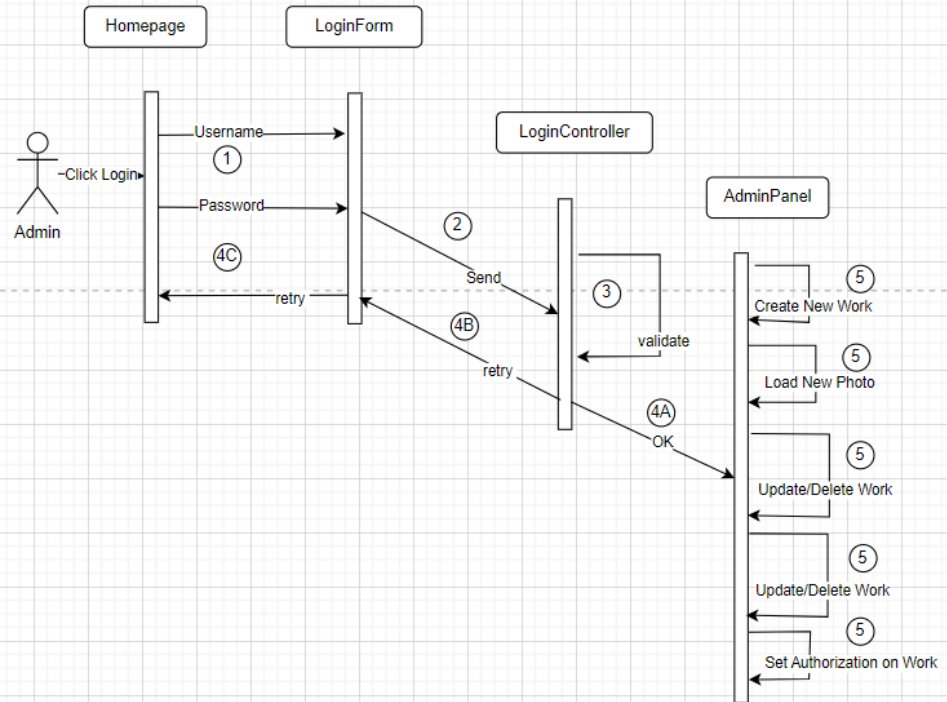
Un utente (Admin o User) dopo essersi registrato ed aver eseguito il login, potrà accedere alla propria galleria di immagini.

Da qui potrà accedere in visualizzazione ai Work e alle relative immagini contenute in essi, per i quali avrà ottenuto l'autorizzazione della visualizzazione.



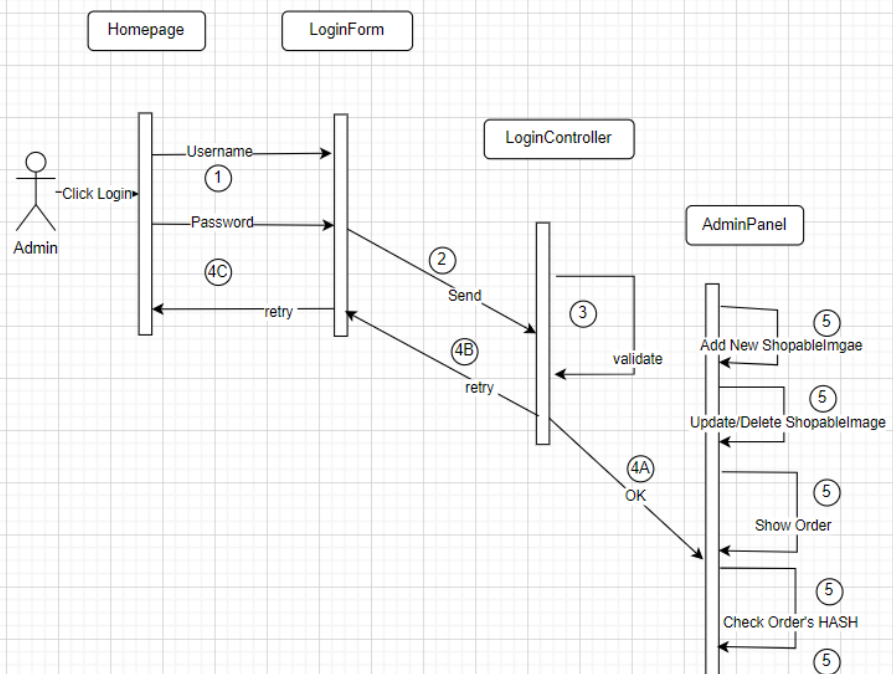
L'admin dal proprio pannello di gestione potrà creare di nuovi Work, avrà la completa gestione di aggiornamento e cancellazione dei Work, potrà caricare immagini specificando a quale Work appartengono. Avrà la completa gestione di aggiornamento e cancellazione delle singole immagini e potrà settare le autorizzazioni di visualizzazione in modo granulare.

L'admin oltre ad avere la visualizzazione completa dei work, potrà crearne di nuovi, caricare immagini, settare le autorizzazioni. Potrà inoltre caricare le Shopable photo sullo store

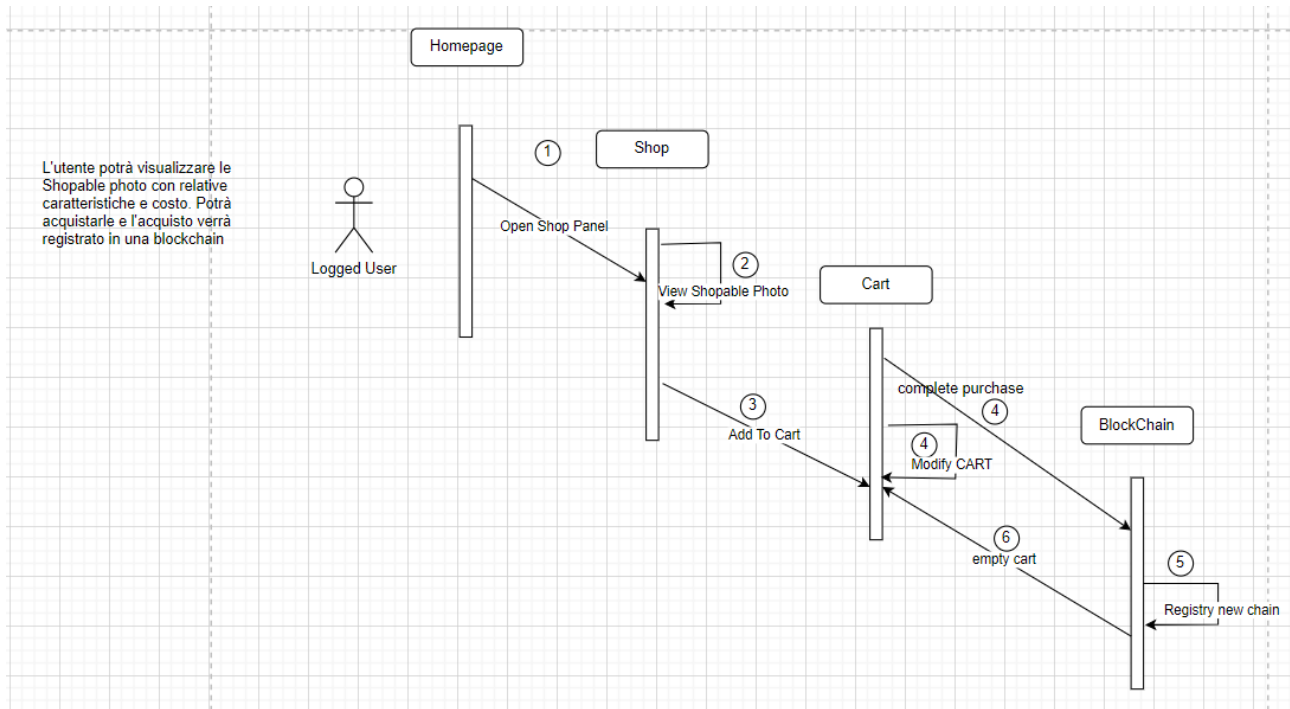


L'admin dalla View AdminShop potrà caricare nuove ShopableImage, cancellare/aggiornare quelle già in vendita, potrà visualizzare tutti gli ordini fatti e verificare la veridicità dell'Hash delle transizioni di vendita.

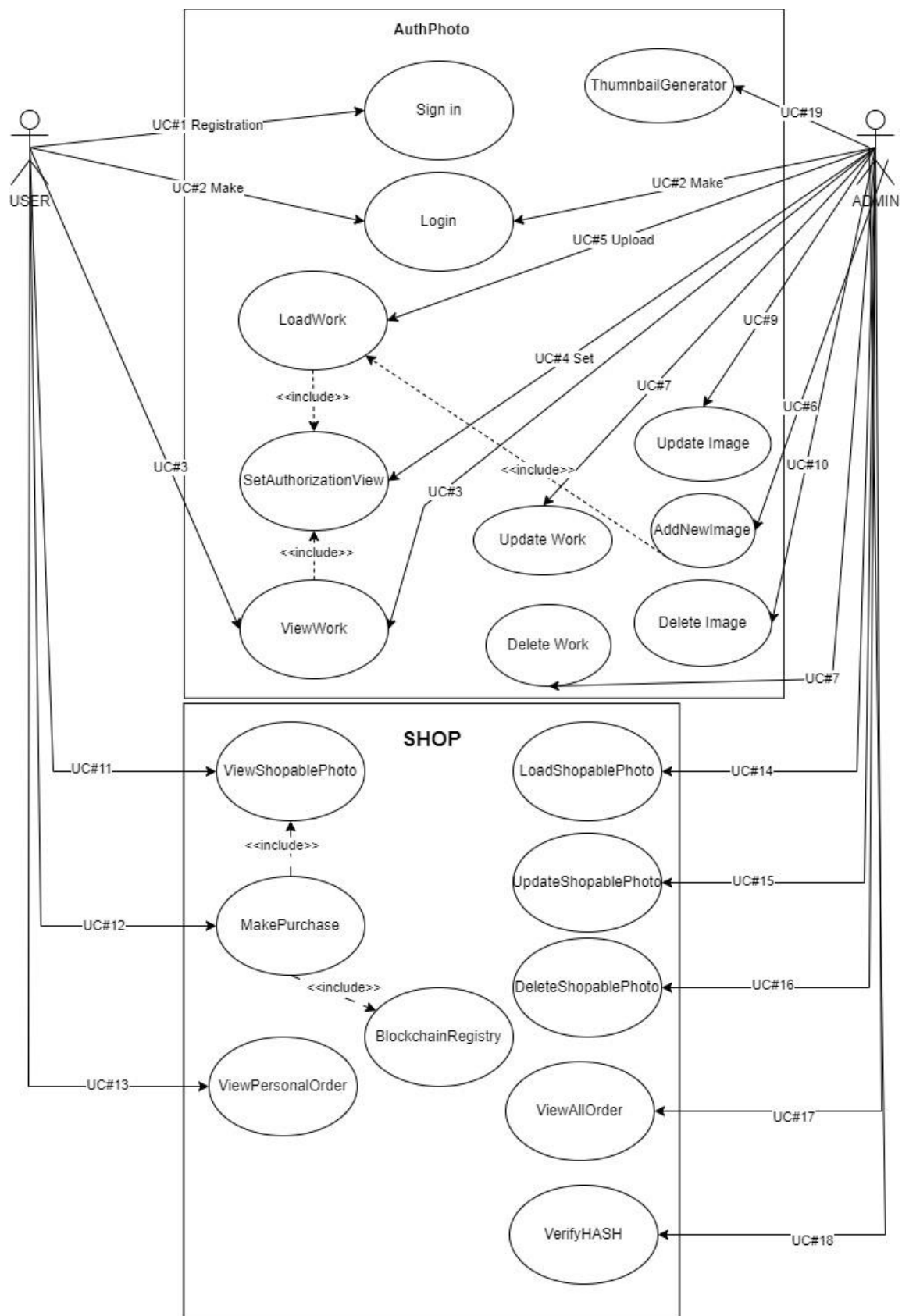
L'admin dalla View AdminShop potrà caricare nuove ShopableImage, cancellare/aggiornare quelle già in vendita, potrà visualizzare tutti gli ordini fatti e verificare la veridicità dell'Hash delle transizioni delle vendite



L'utente potrà visualizzare le ShopableImage con relative caratteristiche e costo. Potrà acquistarle e l'acquisto verrà registrato in una blockchain



Use Case Diagram



- UC#1 Registration: L'utente si registra al sistema.
- UC#2 Make Login: L'utente effettua il login nel sistema.
- UC#3 ViewWork: Gli utenti e l'amministratore accedono in visualizzazione ai Work per i quali ne hanno diritto.
- UC#4 SetAuthorizationView: L'amministratore fornisce le autorizzazioni per la visualizzazione di un Work. In modo speculare l'utente riceve l'autorizzazione per visualizzare un lavoro.
- UC#5 Upload: L'amministratore aggiunge un nuovo Work.
- UC#6 AddNewImage: L'amministratore carica una nuova immagine associando ad un Work
- UC#7 Update Work: L'amministratore aggiorna un Work esistente.
- UC#8 Delete Work: L'amministratore elimina un Work.
- UC#9 Update Image: L'amministratore aggiorna un'immagine esistente.
- UC#10 Delete Image: L'amministratore elimina un'immagine.
- UC#11 ViewShoppablePhoto: L'utente visualizza le foto disponibili per l'acquisto.
- UC#12 MakePurchase: L'utente effettua un acquisto.
- UC#13 ViewPersonalOrder: L'utente visualizza i propri ordini.
- UC#14 LoadShoppablePhoto: L'amministratore carica le foto disponibili per la vendita.
- UC#15 UpdateShoppablePhoto: L'amministratore modifica le foto disponibili per la vendita.
- UC#16 DeleteShoppablePhoto: L'amministratore cancella una o più foto tra quelle disponibili per la vendita.
- UC#17 ViewAllOrder: L'amministratore visualizza tutti gli ordini.
- UC#18 VerifyHASH: L'amministratore verifica l'hash di una transazione nella blockchain.
- UC#19 ThumbnailGenerator: L'amministratore genera le miniature per le immagini contenute in un Work

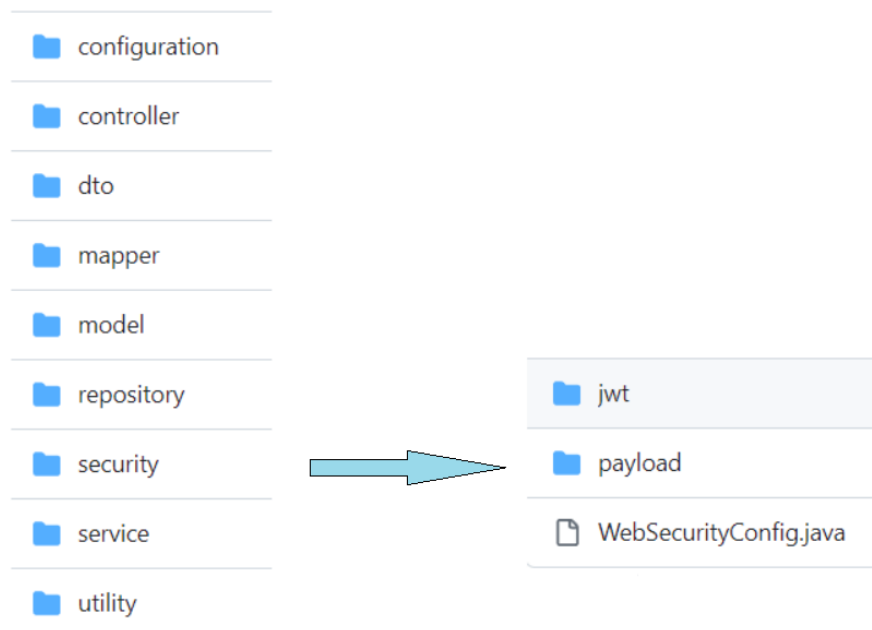
Tabella di copertura dei casi d'uso

	UC#1 Registration	UC#2 Make Login	UC#3 ViewWork	UC#4 SetAuthorizationView	UC#5 Upload	UC#6 AddNewImage	UC#7 Update Work	UC#8 Delete Work	UC#9 Update Image
RF-1.a	X								
RF-1.b		X	X	X					
RF-1.c			X	X					
RF-1.d									
RF-1.e									
RF-1.f									
RF-2.a				X	X	X	X	X	X
RF-2.b					X	X	X	X	X
RF-2.c				X					
RF-3.a					X	X	X		
RF-3.b							X		
RF-3.c									
RF-3.d									
RF-3.e									
RF-3.f									
RF-4.a	X								
RF-4.b				X					
RF-4.c									
RF-4.d		X		X					
RF-4.e		X		X					
RF-5.a									
RF-5.b			X						
RF-6.a									
RF-6.b									
RF-6.c									
RF-6.d									
RF-6.e									
RF-6.f									

	UC#10 Delete Image	UC#11 ViewShopablePhoto	UC#13 MakePurchase	UC#14 ViewPersonalOrder	UC#15 LoadShopablePhoto	UC#16 UpdateShopablePhoto	UC#16 DeleteShopablePhoto	UC#17 ViewAllOrder	UC#18 VerifyHASH	UC#19 ThumbnailGenerator
RF-1.a										
RF-1.b										
RF-1.c										
RF-1.d		X	X		X	X	X	X	X	X
RF-1.e		X	X							
RF-1.f										X
RF-2.a	X									
RF-2.b	X									
RF-2.c										
RF-3.a										
RF-3.b										
RF-3.c										X
RF-3.d										X
RF-3.e										
RF-3.f										X
RF-4.a										
RF-4.b										
RF-4.c										
RF-4.d										
RF-4.e										
RF-5.a										X
RF-5.b										
RF-6.a		X								
RF-6.b		X	X							
RF-6.c								X	X	
RF-6.d								X	X	
RF-6.e					X	X	X			
RF-6.f				X			X			

Descrizione dei package

Prima di analizzare nel dettaglio l'implementazione del back-end con il framework Spring, in questa sezione si analizza la divisione logica dei package.



- **configuration:** contiene la classe utilizzata per eseguire il bootstrap e avviare l'applicazione Spring. Inoltre, contiene il Bean CommandLineAppStartupRunner per l'inizializzazione di uno User e di un Admin.
- **controller:** contiene i controller che si occupano dell'esposizione e nella gestione delle risorse attraverso le operazioni HTTP. Si occupano del mapping delle risorse: gestiscono la conversione dei dati in formato JSON durante la trasmissione delle risposte e l'interpretazione (JSON -> Oggetti del modello dei dati) dei dati ricevuti nelle richieste
- **dto:** contiene i DTO (Data Transfer Object) utilizzati per trasferire dati tra il livello di persistenza dei dati (database) e il livello di esposizione delle API (controller).
- **mapper:** Contiene una classe, che sfruttando i generics, esegue il mapping tra model e DTO e viceversa. Per alcuni parametri che non vengono mappati, la mappatura viene fatta manualmente all'interno dei controller.
- **model:** contiene le classi che modellano che costituiscono il modello di dominio dell'applicazione, la rappresentazione dei concetti chiave e delle entità del dominio del problema.
- **repository:** contiene le classi che sono responsabili della gestione della persistenza dei dati, sviluppate secondo il framework Spring Data JPA.
Le principali responsabilità e funzionalità di questo livello sono: interfacciarsi con il Database, gestire le operazioni CRUD, mappare gli oggetti Java sulle tabelle del database, gestire la conversione dei dati tra i tipi di dati Java e i tipi di dati del database e la gestione della persistenza
- **security:** contiene la Classe WebSecurityConfig che abilita l'autenticazione e l'autorizzazione all'interno dell'applicazione. Inoltre, specifica le configurazioni del sistema di sicurezza per l'applicazione web. Questa classe estende le classi di configurazione fornite da Spring Security per personalizzare il comportamento del sistema di sicurezza.
 - **jwt:** metodi per la creazione, verifica e il controllo del token JWT

- **payload:** DTO per la gestione della login/signin e l'aggiornamento della password di un utente
- **service:** contiene tutte le classi che si trovano tra il layer dei controller e il layer di accesso ai dati (repository/DAO).
- **utility:** contiene due classi di servizio. *ThumbnailGenerator* che si occupa della generazione delle miniature e dell'aggiunta dei watermark. La classe *BlockchainTransactionService* utilizzata per la registrazione di nuove transizioni o della verifica di vecchie transizioni all'interno della blockchain

Implementazione lato JAVA

L'implementazione del progetto è stata realizzata basandosi sull'architettura MVC basata su JakartaEE. In particolare, si è deciso di utilizzare con il framework Spring per la realizzazione del back-end.

Dal "mondo" Spring si è deciso di utilizzare Spring Boot: è un framework open-source per creare applicazioni web basate su Spring, semplificando la configurazione e l'avvio delle applicazioni Spring. La versione utilizzata è la **3.3.2**.

Si è fatto uso del tool <https://start.spring.io> che ci permette di eliminare gran parte della configurazione manuale ed evitare conflitti tra le varie versioni delle librerie esterne.

La community di Spring mette a disposizione anche l'IDE Spring Tool Suite basato sui Eclipse, per questo progetto è stata utilizzata la versione 4.21.0.

Nel dettaglio le tecnologie/librerie utilizzate sono:

- Java OpenJDK 17.0.9
- Maven 3.9.6
- spring-boot-starter-parent 3.24
- spring-boot-starter-data-jdbc
- spring-boot-starter-jpa
- spring-boot-starter-web
- spring-boot-starter-jaxb-runtime (glassfish web server)
- spring-boot-starter-mariadb-java-client
- spring-boot-starter-security
- jjwt-api 0.11.5
- jjwt-impl 0.11.5
- jjwt-jackson 0.11.5
- javax.servlet-api 4.0.1
- Apache Tomcat 10.1.19

Per i test delle API è stato utilizzato PostMan 11.3.2 e Insomnia 10.0.0

L'analisi dell'implementazione percorrerà tutti i package, in modo da analizzare l'architettura generale, ma soffermerà l'attenzione i dettagli implementativi ritenuti più interessanti.

it.myportfolio.configuration.myportfolioApplication.java

Questa, anche se contiene poche righe di codice, è una delle classi principali del progetto. Si occupa di fare il discovery dei vari componenti del bootstrap dell'applicazione.

Nel dettaglio:

- **@SpringBootApplication**: è un'annotazione "scorciatoia" che combina tre annotazioni molto importanti:
 - **@Configuration**: indica che la classe è una sorgente di definizioni di bean per il contesto dell'applicazione.
 - **@EnableAutoConfiguration**: abilita la configurazione automatica di Spring Boot, che tenta di configurare automaticamente i bean necessari per l'applicazione in base alle dipendenze trovate nel classpath.
 - **@ComponentScan**: abilita la scansione dei componenti, permettendo a Spring di trovare e registrare automaticamente i bean all'interno del pacchetto corrente (e/o dei suoi sottopacchetti).
- **@ComponentScan(basePackages = {"it.myportfolio.*"})**: questa annotazione specifica i pacchetti da scansionare per trovare i componenti Spring (come **@Component**, **@Service**, **@Repository**). In questo caso, si specifica al framework Spring di scansionare tutti i sottopacchetti del pacchetto `it.myportfolio`.
- **@EnableJpaRepositories(basePackages = "it.myportfolio.repository")**: questa annotazione abilita l'uso dei repository JPA e il relativo framework di querying Data JPA. In questo caso le interfacce repository sono presenti all'interno del pacchetto `it.myportfolio.repository`.
- **@EntityScan("it.myportfolio.model")**: questa annotazione specifica i pacchetti da scansionare per trovare le entità JPA. In questo caso le entità JPA sono all'interno del pacchetto `it.myportfolio.model`.

Nel codice dello Snippet 1 possiamo vedere la funzione `main()`, che richiama il metodo "padre" di tutta l'applicazione: **`SpringApplication.run(MyApplication.class, args)`**

La chiamata di quel metodo esegue il bootstrap dell'applicazione Spring Boot. Vediamo nel dettaglio cosa succede:

1. Crea un'istanza di **SpringApplication**:

- **SpringApplication** è una classe che esegue automaticamente il bootstrap di un'applicazione Spring da un metodo *main* statico.

2. Configura l'applicazione:

- Determina l'ambiente di esecuzione, se specificato (ad es. `development`, `production`).
- Configura il contesto dell'applicazione (ad es. `ApplicationContext`).
- Abilita il logging, se richiesto.

3. Scansione dei componenti: scansiona i pacchetti per trovare i componenti, le configurazioni e i servizi annotati con **@Component**, **@EnableJpaRepositories**, **@EntityScan**

4. **Auto-configurazione:**

- Configura automaticamente molti aspetti dell'applicazione in base alle dipendenze trovate nel classpath.
- Download e linking automatico di tutto le dipendenze richieste all'interno del pom.xml
- Connessione al database, sfruttando i parametri (credenziali, connettore, dialetto) presenti all'interno del file application.properties

5. **Avvia l'ApplicationContext:** è il container principale di Spring che gestisce i bean e le loro dipendenze.

6. **Avvia il server web integrato:** avvia automaticamente il server web Tomcat incorporato per servire l'applicazione.


```

@SpringBootApplication
@ComponentScan(basePackages = {"it.myportfolio.*"})
@EnableJpaRepositories(basePackages = "it.myportfolio.repository")
@EntityScan("it.myportfolio.model")

public class MyportfolioApplication {

    public static void main(String[] args) throws java.io.IOException {
        SpringApplication.run(MyportfolioApplication.class, args);
    }

    @Component
    public class CommandLineAppStartupRunner implements CommandLineRunner {

        @Autowired
        RoleRepository roleRepository;

        @Autowired
        UserRepository userRepository;

        @Autowired
        WorkRepository workRepository;

        @Autowired
        PasswordEncoder encoder;

        @Override
        public void run(String... args) throws Exception {
            Role userRole = new Role(ERole.ROLE_USER);
            Role adminRole = new Role(ERole.ROLE_ADMIN);
            roleRepository.save(userRole);
            roleRepository.save(adminRole);

            User admin = new User("admin", "admin@test.it", encoder.encode("admin"), "admin",
"admin");
            Set<Role> adminRoles = new HashSet<>();
            adminRoles.add(adminRole);
            adminRoles.add(userRole);
            admin.setRoles(adminRoles);
            admin.setEnable(true);

            userRepository.save(admin);

            User user = new User("user", "user@test.it", encoder.encode("user"), "user", "user");
            Set<Role> userRoles = new HashSet<>();
            userRoles.add(userRole);
            user.setRoles(userRoles);
            user.setEnable(true);
            userRepository.save(user);

            Work work = new Work ();
            work.setCompany("Shop");
            work.setCompletionDate(new java.util.Date());
            work.setTitle("Shop");

            workRepository.save(work);
        }
    }
}

```

Snippet 1: Codice completo classe MyportfolioApplication

All'interno della classe è stato utilizzato un *CommandLineAppStartupRunner* (classe che implementa l'interfaccia *CommandLineRunner*) che viene utilizzato per eseguire del codice subito dopo il completamento del bootstrap dell'applicazione, ma prima che l'applicazione inizi ad accettare richieste. In particolare è stato utilizzato per inizializzare un'utenza con ruolo admin, un'utenza con ruolo user e il Work "Shop" che raccoglierà le immagini acquistabili all'interno dello shop.

Annotations

La prossima sezione descriverà le **entità di dominio** realizzate per modellare il dominio applicativo, ma prima di vederle nel dettaglio, in questa sezione si analizzeranno tutte le annotations utilizzate per definire il comportamento delle entità (modelli), delle tabelle, delle relazioni tra tabelle, e di altri aspetti legati alla gestione della.

- **@Entity**: è utilizzata per indicare che una classe Java rappresenta un'entità persistente, cioè un oggetto che sarà mappato a una tabella nel database relazionale.
Una volta annotata una classe con **@Entity**, ogni istanza di questa classe può essere gestita da JPA/Hibernate come una riga in una tabella del database.
- **@Table(name = "x")** è utilizzata per specificare i dettagli della tabella del database a cui l'entità è mappata. In questo caso è utilizzata per definire il nome della tabella nel database a cui è mappata l'entità.
- **@Table(name = "user", uniqueConstraints = { @UniqueConstraint(columnNames = "username"), @UniqueConstraint(columnNames = "email") })**: oltre a definire il nome della tabella, la **@Table** consente di specificare dei vincoli unici sulle colonne.
In particolare, tramite *uniqueConstraints*, le colonne username e email della tabella user devono essere uniche.
- **@Id**: indica che il campo annotato è la **chiave primaria** dell'entità. Questo campo sarà utilizzato come identificatore univoco per ogni istanza dell'entità.
- **@GeneratedValue(strategy = GenerationType.IDENTITY)**: viene utilizzata insieme a **@Id** per indicare che il valore della chiave primaria deve essere generato automaticamente.
In tutti i modelli ad esclusione dei Model ImageProject e ShopableImage è stato utilizzato il parametro *GenerationType.IDENTITY*, grazie al quale la chiave primaria viene generata dal database attraverso una colonna auto-incrementante.
- **@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)**: viene utilizzata per specificare la strategia di ereditarietà tra entità in Hibernate. Ogni classe concreta della gerarchia ereditaria viene mappata su una propria tabella nel database. Ogni tabella avrà tutte le colonne della classe padre e delle classi figlie.
- **@GeneratedValue(strategy = GenerationType.TABLE)**: la chiave primaria viene generata usando una tabella specifica nel database che tiene traccia dei valori delle chiavi primarie per tutte le entità.
- **@Column(name = "is_sold", columnDefinition = "boolean default false")**: specifica che il campo isSold sarà mappato su una colonna chiamata is_sold nel database e che la colonna sarà di tipo booleano, con false come valore predefinito.
- **@Column(nullable = false)**: specifica che la colonna associata non può contenere valori null e garantisce che il campo abbia sempre un valore prima di essere inserito nel database.
- **@OneToMany(cascade = CascadeType.ALL)** indica una relazione uno-a-molti tra due entità. Il campo *CascadeType* specifica che tutte le operazioni di persistenza (persist, merge, remove, refresh, detach) effettuate sulla entità padre saranno propagate anche alle entità figlie.
- **@JoinColumn(name = "work_id")**: utilizzata per specificare la colonna di join nel database per una relazione tra due entità. Questa annotazione definisce il nome della colonna che fungerà da chiave esterna nella tabella dell'entità figlia.

- **@ManyToOne**: definisce una relazione molti-a-uno tra due entità; viene applicata sul lato "molti" della relazione, cioè sull'entità che contiene la chiave esterna che punta all'entità "uno".
- **@ManyToOne(fetch = FetchType.LAZY)**: definisce una relazione molti-a-molti tra due entità. Tramite il campo *FetchType* si specifica che l'entità correlata deve essere caricata pigramente (solo quando viene effettivamente richiesta). Questo migliora le prestazioni evitando il caricamento di dati non necessari.
- **@JoinTable(name = "visible_work", joinColumns = @JoinColumn(name = "work_id"), inverseJoinColumns = @JoinColumn(name = "user_id"))**: usata per definire la tabella di join in una relazione molti-a-molti.
 - **name**: definisce il nome della tabella di join.
 - **joinColumns**: specifica la chiave esterna nella tabella di join che si riferisce all'entità corrente.
 - **inverseJoinColumns**: specifica la chiave esterna nella tabella di join che si riferisce all'entità opposta.
- **@ManyToOne(mappedBy = "shopableImages")**: indica che questa è la parte inversa di una relazione molti-a-molti e che la gestione della relazione è delegata all'altra entità

it.myportfolio.model

Come visto in precedenza il package model contiene le classi che modellano il dominio applicativo. I model o **entità di dominio (Domain Entities)** rappresentano oggetti del dominio dell'applicazione e sono mappati su tabelle di un database relazionale tramite Hibernate.

Work

Il modello Work rappresenta un progetto fotografico o un insieme di fotografie che condividono una caratteristica comune, come il soggetto (es. paesaggi) o il contesto (es. commissione per un'azienda). All'atto pratico, il Work è un contenitore che raggruppa le fotografie scattate per un cliente specifico o con un obiettivo tematico comune, come la fotografia paesaggistica, commerciale o di eventi.

Il modello Work serve a organizzare e gestire le fotografie in base a progetti o tematiche, facilitando la catalogazione e la presentazione del lavoro svolto in un contesto professionale o artistico.

Le risorse immagini caricate devono poter essere raggruppate per Work ed esistono se e solo se sono collegate ad un Work.

Ogni Work ha come campi:

- **Id**: chiave univoca auto incrementale
- **title**: rappresenta il titolo del lavoro fotografico
- **company**: rappresenta l'azienda/ cliente per il quale il fotografo ha lavoro per realizzare lo shooting. Company può essere utilizzato anche per descrivere un raggruppamento di immagini con caratteristiche precise, ma non appartenenti ad un'azienda (es. fotografia paesaggistica)

- **completionDate:** data in cui è stato completato lo shooting
- **image:** lista di tutte le immagini scattate durante il lavoro fotografico
- **user:** lista di utenti che hanno accesso in visualizzazione a quel lavoro. A livello di database la relazione N-N tra i Work e gli User viene risolta con la tabella intermedia “visible_work”

```
@Entity
@Table(name = "work")
public class Work {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long Id;

    @Column(nullable = false)
    private String title;

    private String company;

    @Column(name = "completion_date")
    private Date completionDate;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "work_id")
    private Set<ImageProject> image;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "visible_work",
        joinColumns = @JoinColumn(name = "work_id"),
        inverseJoinColumns = @JoinColumn(name = "user_id"))
    private Set<User> users = new HashSet<>();

    ...
    //constructor getters and setters
    ...
}
```

Snippet 2: Model Work

ImageProject

Il modello ImageProject rappresenta una singola immagine all'interno di un progetto fotografico. Esso memorizza informazioni fondamentali per l'identificazione e l'accesso a ogni immagine, inclusi un'etichetta descrittiva, l'URL completo dell'immagine e un URL per una versione miniaturizzata.

I campi di ImageProject sono:

- **Id:** chiave univoca auto incrementale
- **Label:** breve descrizione o nome assegnato all'immagine, che può essere utilizzato per identificare o descrivere brevemente il contenuto dell'immagine stessa.
- **URL:** Il campo URL memorizza l'indirizzo (absolute path) completo dell'immagine salvata sul NAS.

- **Thumbnail URL:** memorizza l'indirizzo (absolute path) completo dell'immagine in formato miniatura salvata sul NAS. Questa miniatura viene utilizzata per una visualizzazione più rapida nella galleria di immagini senza dover caricare l'immagine completa.

```
@Entity
@Table(name="image")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class ImageProject {

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    @Column(name="id")
    private Long Id;

    private String label;

    private String URL;

    private String thumbnailURL;

    ...
    //constructor getters and setters
    ...
}
```

Snippet 3: Model ImageProject

ShopableImage

Il modello ShopableImage è un'estensione (sfruttando l'ereditarietà) del modello ImageProject, utilizzato per gestire la vendita di immagini come opere fotografiche uniche. Questo model consente di trattare ogni immagine come un pezzo da collezione, venduto in singola unità.

Una volta venduta, l'immagine non è più disponibile per altri acquirenti, riflettendo il suo carattere esclusivo e da collezione. La relazione con Cart consente ai clienti di aggiungere l'immagine ai loro carrelli per poi eventualmente procedere con l'acquisto, ma la natura unica dell'opera impone che una volta acquistata, essa non possa più essere riacquistata.

I campi di questo model sono:

- **Ereditati da ImageProject:** eredita tutte le proprietà di base dal modello ImageProject, come l'ID, l'etichetta (label), l'URL dell'immagine, e l'URL della miniatura (thumbnailURL).
- **isSold:** Questo campo booleano (isSold) indica se l'immagine è già stata venduta. Dato che ShopableImage rappresenta opere fotografiche in numero unico, una volta che il campo isSold è impostato su true, l'immagine non sarà più disponibile per la vendita.
- **price:** price è un campo obbligatorio che rappresenta il prezzo di vendita dell'immagine.
- **cart:** Questa è una relazione ManyToMany con il modello Cart, che rappresenta i carelli, all'interno dello shop, dei clienti in cui l'immagine è stata aggiunta. Tuttavia, poiché l'immagine è unica, una volta venduta, l'immagine non può essere acquistata: al momento dell'acquisto viene verificato se le immagini contenute nel carrello sono state già comprate, in caso affermativo restituisce un errore contenente quali immagini non possono essere acquistate.

```

@Entity
@Table(name="shopable_image")
public class ShopableImage extends ImageProject{

    @Column(name="is_sold", columnDefinition = "boolean default false")
    private boolean isSold;

    @Column(nullable = false)
    private float price;

    @ManyToMany(mappedBy = "shopableImages")
    private List<Cart> cart;

    ...
    //constructor getters and setters
    ...

```

Snippet 4: Model ShopableImage

User

Il modello User è una rappresentazione completa di un utente all'interno dell'applicazione, che include sia informazioni personali come nome, cognome, username, email e password, sia aspetti legati alla sicurezza e alle autorizzazioni, come lo stato dell'account (enable) e i ruoli (roles).

Inoltre, gestisce le relazioni con i progetti che l'utente può visualizzare (visibleWorks), offrendo una visione chiara e organizzata dei dati utente e delle sue interazioni all'interno del sistema.

I campi di questo model sono:

- **Id:** chiave univoca auto incrementale
- **name, surname:** i campi che memorizzano rispettivamente il nome e il cognome dell'utente. Questi campi servono per identificare l'utente in modo più "umano", oltre che per scopi di visualizzazione.
- **username:** è un campo univoco che identifica l'utente all'interno del sistema. Viene utilizzato per l'autenticazione e deve essere unico per ogni utente, come indicato dal vincolo di unicità (@UniqueConstraint).
- **email:** è un altro campo univoco utilizzato per contattare l'utente. La sua unicità è garantita da un vincolo di unicità (@UniqueConstraint).
- **password:** è il campo che memorizza la password dell'utente in forma crittografata lato DB.
- **enable:** è un campo booleano che indica se l'account dell'utente è attivo (true) o disabilitato (false). Questo campo è utile per gestire lo stato dell'account, ad esempio in caso di sospensione temporanea.
- **VisibleWorks:** è una relazione ManyToMany con il modello Work. Rappresenta i progetti (o insiemi di immagini) che l'utente ha il permesso di visualizzare. Questa relazione è definita come LAZY, il che significa che i Work associati non vengono caricati automaticamente insieme all'utente, ma solo quando richiesto, per migliorare le prestazioni.

- **Roles:** roles è una relazione ManyToMany con il modello Role, che rappresenta i ruoli assegnati all'utente. Questi ruoli definiscono i permessi e le autorizzazioni dell'utente all'interno del sistema, come l'accesso a determinate funzionalità o risorse. La relazione è gestita tramite una tabella di join (user_roles) che collega gli utenti con i ruoli.

```
@Entity
@Table(name = "user", uniqueConstraints = { @UniqueConstraint(columnNames =
"username"),
        @UniqueConstraint(columnNames = "email") })
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String surname;
    private String name;
    private String username;
    private Boolean enable;
    private String email;
    private String password;

    @ManyToMany(mappedBy = "users", fetch = FetchType.LAZY)
    private Set<Work> visibleWorks;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "user_roles", joinColumns = @JoinColumn(name = "user_id"),
inverseJoinColumns = @JoinColumn(name = "role_id"))
    private Set<Role> roles = new HashSet<>();

    ...
    //constructor getters and setters
    ...
}
```

Snippet 5: Model User

ERole

ERole è un'enumerazione (enum) che definisce i ruoli disponibili all'interno del sistema.

- **ROLE_USER:** Rappresenta un ruolo standard per gli utenti dell'applicazione. Gli utenti con questo ruolo hanno accesso alle gallerie per i quali hanno diritto di visualizzazione e allo shop.
- **ROLE_ADMIN:** Rappresenta un ruolo amministrativo con privilegi elevati. Gli utenti con questo ruolo hanno accesso a tutte le funzionalità dell'applicazione: gestione e amministrazione Work e immagini, la gestione degli utenti.

```
public enum ERole {
    ROLE_USER, ROLE_ADMIN
}
```

Snippet 6: Dettaglio enum ERole

Role

Il modello Role rappresenta un'entità nel database che memorizza le informazioni dei due ruoli definiti all'interno dell'applicazione. Questi ruoli sono utilizzati per controllare e gestire le autorizzazioni e i permessi degli utenti.

```
@Entity
@Table(name = "roles")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Enumerated(EnumType.STRING)
    @Column(length = 20)
    private ERole name;

    ...
    //constructor, getters and setters
    ...
}
```

Snippet 7: Model Role

Cart

Il modello Cart gestisce i carrelli della spesa per gli utenti, consentendo loro di selezionare e salvare immagini acquistabili che intendono comprare. Ogni carrello è associato a un singolo utente e può contenere più immagini ShopableImage.

I campi di questo model sono:

- **Id:** chiave univoca auto incrementale
- **User:** rappresenta l'utente associato a questo carrello. La relazione è definita come @OneToOne, indicando che ogni carrello è associato a un solo utente e viceversa.
- **ShopableImages:** è una relazione ManyToMany che rappresenta le immagini acquistabili aggiunte al carrello. Utilizza la tabella di join "cart_shopable_image" per gestire l'associazione tra i carelli degli utenti e le immagini.


```

@Entity
public class Cart {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @OneToOne
    private User user;

    @ManyToMany
    @JoinTable(
        name = "cart_shopable_image",
        joinColumns = @JoinColumn(name = "cart_id"),
        inverseJoinColumns = @JoinColumn(name = "shopable_image_id")
    )
    private List<ShopableImage> shopableImages;

    ...
    //constructor, getters and setters
    ...
}

```

Snippet 8: Model Cart

SalesOrder

Il modello SalesOrder è essenziale per gestire gli ordini di vendita all'interno del sistema, includendo non solo le informazioni di base sull'acquisto, come la data e l'utente associato, ma ha anche il compito di memorizzare l'hash registrato all'interno della blockchain. Il campo hash, viene generato tramite uno script che permette di registrare l'ordine su un fork di Ethereum. Questo permette di fornire ad entrambi le parti (acquirente e venditore) un ulteriore livello di sicurezza e trasparenza, assicurando che ogni ordine sia verificabile e immutabile, proteggendo così gli interessi sia dell'acquirente che del venditore.

Vedremo più avanti una visione più dettagliata del meccanismo di registrazione all'interno della blockchain.

I campi di questo model sono:

- **Id:** chiave univoca auto incrementale
- **Timestamp:** campo di tipo Date che memorizza la data e l'ora in cui l'ordine è stato effettuato.
- **PurchasedImage:** relazione OneToMany con il modello ShopableImage. Questo campo rappresenta la lista delle immagini acquistate come parte dell'ordine.
- **User:** rappresenta l'utente che ha effettuato l'ordine. La relazione è definita come ManyToOne, indicando che più ordini possono essere associati a un singolo utente, ma ogni ordine è legato a un solo utente.
- **Hash:** hash è un campo di tipo String che memorizza un valore di hash unico associato all'ordine e generato tramite le chiamate API fornite dalla libreria Infura.

```

@Entity
@Table(name = "sales_order")
public class SalesOrder {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long Id;

    private Date timestamp;

    @OneToMany
    @JoinColumn(name = "sales_order_id")
    private List<ShopableImage> purchasedImage;

    @ManyToOne
    private User user;

    private String hash;

    ...
    //constructor, getters and setters
    ...
}

```

Snippet 9: Model SalesOrder

it.myportfolio.dto

Un **DTO (Data Transfer Object)** è un oggetto usato per trasportare dati tra i livelli esposti al pubblico (in questo caso le API esposte tramite i controller) e il livello di persistenza (un database).

I DTO sono usati principalmente per:

1. **Evitare di esporre oggetti di dominio direttamente:** esporre direttamente questi oggetti nelle API può portare a problemi di sicurezza e manutenibilità. I DTO offrono un livello di astrazione che consente di filtrare e controllare quali dati vengono effettivamente esposti all'esterno.
2. **Ridurre il sovraccarico delle comunicazioni:** i DTO consentono di inviare solo i dati necessari, riducendo la quantità di informazioni che viaggiano sulla rete.
3. **Separazione dei concetti:** I DTO separano la logica di business dalle informazioni di presentazione, migliorando la manutenibilità del codice. Le entità del dominio possono contenere logica di business e associazioni complesse, mentre i DTO contengono solo dati necessari per una specifica operazione o visualizzazione.
4. **Sicurezza dei dati:** esporre le entità del dominio direttamente può essere pericoloso. Ad esempio, un'entità potrebbe contenere campi sensibili come password o altre informazioni che non dovrebbero essere inviate al client.

5. **Aggiornamento:** i DTO possono aiutare a gestire diverse versioni dell'API. Ad esempio, se una nuova versione di un'API richiede nuovi campi o una struttura di dati diversa, è possibile creare nuovi DTO specifici per la nuova versione, mantenendo la compatibilità con le versioni precedenti.

NOTA: i DTO non hanno il compito di gestire la persistenza all'interno dei database, per questo non hanno l'annotazione @Entity.

Nella sezione precedente sono stati descritti nel dettaglio tutti i model presenti nel progetto, questa parte si concentrerà sugli aspetti più rilevanti e sui punti di maggiore interesse relativi ai DTO, si metteranno in evidenza solo le caratteristiche e le implementazioni che meritano particolare attenzione.

UserPersonalDetailsDTO vs User

In questa sezione si mette a confronto il Model User ed il DTO UserPersonalDetailsDTO per mettere in risalto le caratteristiche di entrambi e capirne le differenze e i differenti usi.

1. Differenze di Scopo e Ruolo

- **Model (User):** è la rappresentazione dell'utente fisico all'interno del database. Include tutti i campi che corrispondono alle colonne della tabella del database, contiene anche le relazioni con altre entità (ad esempio, roles e visibleWorks). Questo oggetto viene utilizzato nelle operazioni di persistenza e recupero dei dati.
- **DTO (UserPersonalDetailsDTO):** il DTO contiene solo i campi necessari per un particolare scopo o contesto, riducendo così la quantità di dati trasferiti e migliorando l'efficienza. L'esempio presentato non riporta i campi come username, enable, password e visibleWorks, includendo solo i dati rilevanti per l'uso specifico (come name, surname, email, role). Questo DTO ad esempio è utilizzato come ritorno dell'API GET "/api/user" per restituire l'anagrafica dell'utente per un eventuale aggiornamento dei campi, permette di nascondere dettagli sensibili o irrilevanti del modello, come la password o il campo username. Questo riduce il rischio di esposizione accidentale di informazioni sensibili.

```

public class UserPersonalDetailsDTO {

    private Long Id;
    private String surname;
    private String name;
    private String email;
    private Set<Role> role;

    ...
    //getters and setters
    ...
}

@Entity
@Table(name = "user", uniqueConstraints = { @UniqueConstraint(columnNames =
"username"),
        @UniqueConstraint(columnNames = "email") })
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String surname;
    private String name;
    private String username;
    private Boolean enable;
    private String email;
    private String password;

    @ManyToMany(mappedBy = "users", fetch = FetchType.LAZY)
    private Set<Work> visibleWorks;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name = "user_roles", joinColumns = @JoinColumn(name = "user_id"),
inverseJoinColumns = @JoinColumn(name = "role_id"))
    private Set<Role> roles = new HashSet<>();

    ...
    //getters and setters
    ...
}

```

Snippet 10: dettaglio DTO UserPersonalDetailsDTO

SignupRequest

Un altro esempio di utilizzo di un DTO è durante il processo di registrazione di un utente. Il DTO SignupRequest è progettato per gestire i dati necessari durante il processo di registrazione di un nuovo utente nell'applicazione. Questo oggetto è utilizzato per raccogliere e trasferire in modo sicuro le informazioni dal client (frontend) al backend, dove verrà gestita la creazione dell'utente e la persistenza sul database.

Il DTO permette di limitare l'esposizione dei campi sensibili e di controllare quali dati vengono accettati dall'esterno. Ad esempio, nel processo di registrazione, il DTO accetta solo i campi

strettamente necessari, evitando che informazioni non desiderate o potenzialmente pericolose vengano iniettate.

```
public class SignupRequest {  
    private String username;  
    private String email;  
    private String surname;  
    private String name;  
    private String password;  
  
    ...  
    //getters and setters  
    ...  
}
```

Snippet 11: DTO SignupRequest utilizzato durante la registrazione di un nuovo utente

DetailsSalesOrderDTO

Vediamo adesso un altro uso dei DTO, ovvero quello di dare in output valori che non sono presenti all'interno del DB, ma vengono calcolati/generati a runtime al momento della richiesta.

Il DetailsSalesOrderDTO è un DTO che viene utilizzato per fornire agli utenti Admin una rappresentazione dettagliata di un ordine di vendita. Questo DTO viene utilizzato per trasferire al client tutte le informazioni rilevanti su un ordine di vendita, rendendo disponibili non solo i dati presenti nel modello SalesOrder, ma anche alcune informazioni calcolate dinamicamente.

Un aspetto significativo del DetailsSalesOrderDTO è che i campi totalPrice e piece non sono memorizzati direttamente nel modello SalesOrder. Invece, questi valori vengono calcolati dinamicamente ogni volta che il DTO viene richiesto.

Vediamo i campi più interessanti:

- **purchaseImage:** Una lista di oggetti ShopableImageDTO che rappresentano le immagini associate agli articoli acquistati.
- **totalPrice:** Questo campo rappresenta il prezzo totale dell'ordine, ossia la somma dei prezzi di tutti gli articoli acquistati.
- **piece:** Indica il numero totale di pezzi acquistati nell'ordine.

Grazie a questo oggetto si evitano di memorizzare dati calcolati e superflui che possono essere derivati nel database, riducendo la ridondanza e semplificando la struttura del modello.

I dati vengono generati solo quando effettivamente necessari, riducendo così il rischio di inconsistenze.

```

public class DetailsSalesOrderDTO {

    private Long Id;
    private Date timestamp;
    private List<ShopableImageDTO> purchaseImage;
    private Float totalPrice;
    private String username;
    private int piece;
    private String hash;

    ...
    //getters and setters
    ...
}

```

Snippet 12: Dettaglio DetailsSalesOrderDTO

SimpleShopableImageDTO

Per ultimo vediamo l'esempio di semplice, ma intuitivo dei DTO. Ovvero la semplificazione del modello di Dominio, in un oggetto più semplice da utilizzare per passare informazioni nelle due direzioni backend -> frontend e frontend-> backend.

```

public class SimpleShopableImageDTO {

    private Long Id;
    private String label;
    private float price;

    ...
    //getters and setters
    ...
}

```

Snippet 13: Dettaglio SimpleShopableImageDTO

it.myportfolio.mapper

All'interno di questo package è presente la classe Mapper, che sfruttando i generics, esegue il mapping tra model e DTO e viceversa. Per alcuni parametri che non vengono mappati, la mappatura viene fatta manualmente all'interno dei controller.

Il metodo *toEntity* utilizza BeanUtils (libreria Spring Framework) e la riflessione per copiare le proprietà con lo stesso nome e tipo dal DTO all'oggetto Entity. Praticamente, se DTO e Entity hanno campi comuni (ad esempio id, Url ecc), i valori verranno copiati dall'oggetto DTO all'oggetto entity.

Nel dettaglio quello che viene fatto:

- Tramite **entityClass.getConstructor()** viene recuperato il costruttore predefinito (senza parametri) della classe entityClass.
- Il metodo **constructor.newInstance()** crea una nuova istanza di Entity utilizzando il costruttore recuperato.

- Tramite il metodo statico `copyProperties` della classe `BeanUtils` si copiano i valori delle proprietà del bean sorgente specificato nel bean di destinazione.

```
public static <Entity, DTO> Entity toEntity(Class<Entity> entityClass, DTO dto)
{
    Entity entity = null;
    try {
        Constructor<Entity> constructor = entityClass.getConstructor();
        entity = constructor.newInstance();
        BeanUtils.copyProperties(dto, entity);
    } catch (InstantiationException | IllegalAccessException |
NoSuchMethodException
        | InvocationTargetException e) {
        e.printStackTrace();
    }
    return entity;
}
```

Snippet 14: Dettaglio Mapper Generics DTO->entity

Il metodo `toDTO` è il complemento del metodo `toEntity`. Serve a convertire un oggetto di tipo `Entity` (un'entità del dominio) in un oggetto di tipo `DTO` (Data Transfer Object).

```
public static <Entity, DTO> DTO toDTO(Class<DTO> dtoClass, Entity entity) {
    DTO dto = null;
    try {
        Constructor<DTO> constructor = dtoClass.getConstructor();
        dto = constructor.newInstance();
        BeanUtils.copyProperties(entity, dto);
    } catch (InstantiationException | IllegalAccessException |
NoSuchMethodException
        | InvocationTargetException e) {
        e.printStackTrace();
    }
    return dto;
}
```

Snippet 15: Dettaglio Mapper Generics entity->DTO

it.myportfolio.service

Dopo aver visto i modelli di dominio, gli elementi che ci permettono di passare informazioni tra client e server e gli oggetti che si occupano di fare il mapping, vediamo adesso il layer Service.

Il Service layer è uno dei layer architettonici fondamentali, principalmente responsabile dell'implementazione della Business Logic di un'applicazione. Si trova tra il controller e il data access layer (repository o DAO).

Grazie all'aggiunta di questo layer otteniamo i seguenti vantaggi:

- **Inversion of Control (IoC):** Spring gestisce l'istanziatura e l'iniezione dei service (tramite Dependency Injection). Non dobbiamo creare manualmente gli oggetti dei service, ma possono essere iniettati tramite l'annotazione come `@Autowired`.
- **Riuso:** essendo separati dal livello di presentazione e accesso ai dati, possono essere facilmente riutilizzati in altre parti dell'applicazione o in altri progetti.

- **Loose Coupling:** utilizzando interfacce per classi di servizio e iniettandole tramite l'iniezione di dipendenza di Spring, si promuove un loose coupling all'interno dell'applicazione, che la rende più manutenibile e testabile.
- **Separation of concerns:** avere un livello di servizio separato assicura una chiara separazione delle preoccupazioni. Il livello controller/web si concentra sulla gestione delle richieste e delle risposte HTTP, mentre il livello di servizio si concentra sulla Business Logic. Questa separazione rende la base di codice organizzata e più facile da gestire.

Il livello Service collabora con il livello di accesso ai dati (repository o DAO), che vedremo nella prossima sezione, per recuperare, manipolare e archiviare i dati.

Annotando una classe con `@Service`, la si rende un bean gestito dal container di Spring. Questo significa che Spring si occuperà di creare e gestire l'istanza di questa classe, oltre a gestire le sue dipendenze tramite dependency injection. Grazie a questa annotazione è possibile iniettare il service in altre parti dell'applicazione, come controller o altri service, senza dover creare manualmente istanze della classe.

A titolo di esempio analizziamo il *WorkService* ed il metodo *addToCart* del *CartService*.

Il service *WorkService* presentato implementa una serie di operazioni CRUD (Create, Read, Update, Delete) per gestire l'entità *Work*. Questa classe utilizza il repository *WorkRepository* per interagire con il database e contiene la logica necessaria per eseguire operazioni di base sui dati.

Create (C) - Aggiunta di un nuovo elemento: l'operazione di creazione viene gestita dal metodo *addWork*. Questo metodo riceve un oggetto *Work*, lo salva nel database tramite *workRepository.save(work)* e restituisce l'oggetto salvato, che rappresenta l'elemento appena creato con tutte le informazioni aggiornate (incluso, l'ID generato automaticamente).

Read (R) - Lettura di uno o più elementi: Ci sono vari metodi che implementano la funzionalità di lettura:

- **getWorkById(Long id):** Restituisce un oggetto *Work* avvolto in un *Optional* basato sull'ID fornito, se esiste.
- **getAllWork():** Restituisce una lista di tutti gli oggetti *Work* presenti nel database.
- **getWorkDTOByIdAndUser(Long workId, Long userId):** Questo metodo è una lettura che restituisce un oggetto *Work* specifico in base all'ID del lavoro e all'ID dell'utente, utilizzando un metodo personalizzato definito nel repository. Permette di leggere dal Database un *Work* e allo stesso tempo di verificare che l'utente ne abbia i diritti di visualizzazione

Update (U) - Aggiornamento di un elemento esistente: Il metodo *updateWork* gestisce l'aggiornamento di un oggetto *Work* esistente. Prende come input l'ID dell'oggetto da aggiornare e un oggetto *Work* contenente i nuovi dati. Se l'elemento esiste, ne aggiorna i campi (titolo, azienda, data di completamento) e lo salva nuovamente nel database.

Delete (D) - Eliminazione di un elemento: Il metodo *deleteWorkById* gestisce l'eliminazione di un oggetto *Work* dal database. Riceve come parametro l'ID dell'oggetto da eliminare e utilizza il repository per rimuovere l'elemento corrispondente.


```

@Service
public class WorkService {

    @Autowired
    private WorkRepository workRepository;

    public Optional<Work> getWorkById(Long id) {
        return workRepository.findById(id);
    }

    public Work addWork(Work work) {
        return workRepository.save(work);
    }

    public void deleteWorkById(Long id) {
        workRepository.deleteById(id);
    }

    public List<Work> getAllWork() {
        return workRepository.findAll();
    }

    public Work updateWork(Long id, Work updatedWork) {
        Work existingWork = workRepository.findById(id).orElse(null);
        if (existingWork == null) {
            return null;
        }

        existingWork.setTitle(updatedWork.getTitle());
        existingWork.setCompany(updatedWork.getCompany());
        existingWork.setCompletionDate(updatedWork.getCompletionDate());

        return workRepository.save(existingWork);
    }

    public Work getWorkDTOByIdAndUser(Long workId, Long userId) {
        Optional<Work> optionalWork =
workRepository.findWorkByIdAndUserId(workId, userId);

        if (optionalWork.isPresent()) {
            return optionalWork.get();
        } else {
            return null;
        }
    }
}

```

Snippet 16: Service Layer Work

Nell'esempio precedente abbiamo visto l'uso di metodi tutti esposti dai DAO senza l'uso di codice ausiliario. Vediamo adesso il metodo *addToCart*

Questo metodo permette di aggiungere un oggetto *ShopableImage* al carrello di un utente.

1. Recupero del carrello dell'utente:

- Il metodo cerca il carrello associato all'utente (user) utilizzando il metodo *cartRepository.getCartByUser(user)*.

- Se il carrello non esiste, viene creato un nuovo carrello per l'utente, associato e salvato nel database.

2. Recupero dell'oggetto ShopableImage:

- Viene recuperato l'oggetto ShopableImage corrispondente all'ID fornito (shopableImageId) dal repository shopableImageRepository.

3. Aggiornamento del carrello:

- Viene nuovamente recuperato il carrello dell'utente e si verifica se esiste già una lista di immagini (List<ShopableImage> images) associata al carrello.
- Se la lista è null, viene creata una nuova lista, l'immagine viene aggiunta alla lista, e la lista viene associata al carrello.
- Se la lista esiste, l'immagine viene semplicemente aggiunta alla lista e il carrello viene aggiornato.

4. Aggiornamento del lato ShopableImage:

- Viene gestita la relazione bidirezionale tra Cart e ShopableImage.
- Si controlla se l'immagine ha già una lista di carrelli associati (List<Cart> SHCarts). Se la lista è null, ne viene creata una nuova e il carrello corrente viene aggiunto. Se la lista esiste, il carrello viene semplicemente aggiunto alla lista.

5. Salvataggio finale:

- L'immagine e il carrello aggiornati vengono salvati nel database tramite i rispettivi repository, mantenendo così la consistenza delle relazioni tra le entità.
- Il metodo restituisce l'oggetto Cart aggiornato.

```

public Cart addToCart(User user, Long shopableImageId) {

    Optional <Cart> optionalCart = cartRepository.getCartByUser(user);

    if (optionalCart.isEmpty()) {
        Cart cartToSave = new Cart();
        cartToSave.setUser(user);
        cartRepository.save(cartToSave);
    }

    ShopableImage shopableImage =
        shopableImageRepository.findById(shopableImageId).get();

    //aggiornamento lato CART
    Cart DBcart = cartRepository.getCartByUser(user).get();
    List<ShopableImage> images = DBcart.getImages();
    if (images == null) {
        List<ShopableImage> image = new ArrayList<>();
        image.add(shopableImage);
        DBcart.setImages(image);
    }
    else {
        images.add(shopableImage);
        DBcart.setImages(images);
    }

    //aggiornamento lato shopableImage
    List<Cart> SHCarts = shopableImage.getCart();
    if (SHCarts == null) {
        List<Cart> SHCart = new ArrayList<>();
        SHCart.add(DBcart);
        shopableImage.setCart(SHCart);
    }
    else {
        SHCarts.add(DBcart);
        shopableImage.setCart(SHCarts);
    }

    shopableImageRepository.save(shopableImage);

    return cartRepository.save(DBcart);

}

```

Snippet 17: Dettaglio aggiunta Image alla Carrello dell'utente

it.myportfolio.repository

In questa sezione vedremo i repository, che sono componenti fondamentali utilizzati per interagire con il database. Si basano sul concetto di **DAO (Data Access Object)**, che fornisce un'interfaccia per la persistenza dei dati e per le operazioni di accesso ai dati (come CRUD: Create, Read, Update, Delete).

Spring Data JPA

L'ambiente Spring mette a disposizione il modulo Spring Data che permette di semplificare lo stato di persistenza rimuovendo completamente l'implementazione dei DAO dall'applicazione, in modo da semplificare l'interazione con i database utilizzando Java Persistence API (JPA). Per fare ciò, l'interfaccia DAO deve estendere JpaRepository in modo tale che Spring Data creerà automaticamente un'implementazione dotata dei metodi CRUD più rilevanti per l'accesso ai dati.

L'obiettivo principale di Spring Data JPA è ridurre la quantità di codice boilerplate necessario per interagire con il database. Spring Boot consente di definire repository utilizzando interfacce, senza dover implementare manualmente i metodi di accesso ai dati, detti CRUD (Create, Read, Update, Delete) e query personalizzate.

Fornisce strumenti per:

- Creare facilmente repository di dati per gestire entità JPA.
- Effettuare operazioni CRUD e di query in modo semplice.
- Automatizzare l'implementazione di metodi di accesso ai dati tramite l'uso di interfacce e naming convention.
- Supportare query personalizzate sia con JPQL (Java Persistence Query Language) che con SQL nativo.

La configurazione di Spring Data JPA è molto semplice, attraverso il file *application.properties* è possibile configurare le proprietà per la connessione al database:

```
spring.datasource.url=jdbc:mariadb://localhost:3306/myportfolio-db
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=org.mariadb.jdbc.Driver
spring.jpa.hibernate.ddl-auto=create
#spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
```

Snippet 18: Configurazione Datasource

JpaRepository

La classe *org.springframework.data.jpa.repository.JpaRepository* è una delle interfacce di Spring Data JPA che fornisce un'implementazione generica del pattern **Repository**. Questa interfaccia estende altre interfacce più semplici come *CrudRepository* e *PagingAndSortingRepository*. Grazie ad essa è possibile effettuare operazioni di persistenza senza scrivere codice SQL o implementare manualmente le query.

JpaRepository è un'interfaccia generica con due parametri:

```
public interface JpaRepository<T, ID> extends PagingAndSortingRepository<T, ID>
{
}
}
```

Dove:

- **T**: Rappresenta il tipo dell'entità (model) che vogliamo gestire.
- **ID**: Rappresenta il tipo dell'identificatore primario dell'entità (ad esempio, Long).

Questa interfaccia dispone di alcuni metodi CRUD più importanti disponibili (che vedremo nella prossima sezione), ma offre anche la possibilità di scrivere query in linguaggio JPQL e l'uso del modulo Query Method: consente di definire metodi di query nel repository semplicemente dichiarando il nome del metodo secondo una convenzione specifica, senza dover scrivere implementazione.

TIPS: PER IMPOSTAZIONE PREDEFINITA, I METODI EREDITATI DA `CRUDRepository` EREDITANO LA CONFIGURAZIONE TRANSAZIONALE DA `SimpleJpaRepository`, IN PARTICOLARE:

- I **METODI DI LETTURA** (COME `findById`, `findAll`, ETC.) SONO CONSIDERATI READ-ONLY. SPRING AGGIUNGE AUTOMATICAMENTE L'ANNOTAZIONE `@Transactional(readOnly = true)` A QUESTI METODI.
- I **METODI DI SCRITTURA** (COME `save`, `delete`, `deleteById`, ETC.) SONO TRANSAZIONALI: LE OPERAZIONI DI SCRITTURA VENGONO ESEGUITE ALL'INTERNO DI UNA TRANSAZIONE. SE QUALCOSA VA STORTO DURANTE L'ESECUZIONE DEL METODO, LA TRANSAZIONE VIENE ANNULLATA (ROLLBACK) E NESSUNA MODIFICA VIENE APPLICATA AL DATABASE.

Metodi CRUD

Spring Data JPA fornisce già una serie di metodi CRUD (Create, Read, Update, Delete) che coprono la maggior parte delle necessità comuni di manipolazione attraverso le sue interfacce, come **`CrudRepository`**, **`PagingAndSortingRepository`**, e **`JpaRepository`**. Questi metodi semplificano le operazioni di base per la gestione delle entità nel database, senza la necessità di scrivere codice aggiuntivo.

Esempi di metodi disponibili sono:

- `deleteById`
- `existsById`
- `findById`
- `save`
- `findAll`
- `count`
- `exists`

Query Methods

Grazie a questa funzionalità è possibile definire metodi che eseguiranno query SQL sul DB semplicemente dichiarando il nome del metodo secondo una convenzione specifica. Spring Data JPA genera automaticamente una query basata sui campi dell'entità per eseguire operazioni di ricerca senza la necessità di scrivere query SQL o JPQL manualmente.

Spring Data JPA supporta molte keyword, le più comuni sono:

- **findBy:** Seleziona entità in base a una condizione.
- **countBy:** Conta il numero di entità che soddisfano una condizione.
- **deleteBy:** Elimina entità che soddisfano una condizione.
- **existsBy:** Verifica se esistono entità che soddisfano una condizione.

Si possono anche combinare queste keyword con operatori logici e confronti:

- **And, Or:** Operatori logici per concatenare condizioni.
- **GreaterThan, LessThan:** Confronti numerici.
- **Between:** Filtra un campo tra due valori.
- **Like:** Cerca valori che corrispondono a un pattern.
- **In:** Cerca valori all'interno di una lista.
- **OrderBy:** Ordina i risultati per uno o più campi.

I Query Methods sono ideali per query semplici e comuni, in alternativa è possibile utilizzare l'annotazione @Query per scrivere query JPQL o SQL personalizzate.

ImageRepository

Questo il più semplice repository del progetto. Come possiamo notare non sono stati dichiarati ulteriori metodi, in quanto quelli già inclusi erano sufficienti per il model ImageProject.

```
@Repository
public interface ImageRepository extends JpaRepository<ImageProject, Long> {
}
```

Snippet 19: Repository ImageProject

WorkRepository

In questo repository possiamo notare un esempio di query scritta in JPQL per il recupero di un Work dato un Id che hanno un legame di join con un ID di un User, anch'esso passato come parametro. In altre parole stiamo verificando che la richiesta di visualizzazione di un Work da parte di un utente sia legittima.

```
@Repository
public interface WorkRepository extends JpaRepository<Work, Long> {

    @Query("SELECT w FROM Work w JOIN w.users u WHERE w.id = :workId AND u.id = :userId")
    Optional<Work> findWorkByIdAndUserId(@Param("workId") Long workId,
    @Param("userId") Long userId);

}
```

Snippet 20: Repository Work

UserRepository

In questo repository possiamo vedere un metodo creato tramite annotazione @Query per il recupero di tutti i Work visualizzabili da un certo User.

Sotto possiamo vedere 4 Query method per:

- Recupero di tutti i campi di un User fornito lo username
- Verifica della presenza di un User con un certo username passato come parametro
- Verifica della presenza di un User con una certa mail passata come parametro
- Recupero di tutti i campi di un User passando il suo username e verificando nello stesso tempo che quell'utenza abbia il campo Enable a true.

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    @Query("SELECT w FROM User u JOIN u.visibleWorks w WHERE u.id = :userId")
    public Set<Work> findVisibleWorksByUserId(@Param("userId") Long userId);

    Optional<User> findByUsername(String username);

    Boolean existsByUsername(String username);

    Boolean existsByEmail(String email);

    Optional<User> findByUsernameAndEnable(String username, boolean enable);

}
```

Snippet 21: Repository User

it.myportfolio.controller

Ultimiamo la sezione dei vari componenti implementati in questo progetto, con i componenti che si occupano dell'interazione Client-Server: i controller.

Il ruolo del Controller è quello di gestire le richieste HTTP (GET, POST, PUT, DELETE, PATCH), elaborarle e restituire una risposta appropriata.

In Spring Boot per "eleggere" una classe java a Controller (con architettura API RESTful) basta annotarla con @RestController: un'annotazione specializzata di @Controller che combina anche l'annotazione @ResponseBody, in modo da far restituire direttamente i dati nel corpo della risposta (solitamente in formato JSON o XML), invece di restituire una vista.

```

@CrossOrigin(origins = "*", maxAge = 3600)
@RestController
@RequestMapping("/api/work")
public class WorkController {

    @Autowired
    WorkService workService;

    @Autowired
    private UserService userService;

    @Autowired
    JwtUtils jwtUtils;

    @PreAuthorize("hasRole('USER') or hasRole('ADMIN')")
    @GetMapping("/mywork")
    public ResponseEntity<Set<WorkDTO>>
getVisibleWorksByUserId(HttpServletRequest request) {
        String token = jwtUtils.getJwtFromCookies(request);
        if (jwtUtils.validateJwtToken(token)) {
            Long userId = (Long) jwtUtils.getUserIdFromJwtToken(token);
            Set<Work> works = userService.findVisibleWorksByUserId(userId);
            Set<WorkDTO> workDtos = new HashSet<>();

            for (Work work : works) {
                workDtos.add(WorkDTO.fromWork(work));
            }

            if (workDtos.isEmpty()) {
                return ResponseEntity.notFound().build();
            }

            return ResponseEntity.ok(workDtos);

        } else {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
        }
    }
    .....
}

```

Snippet 22: Dettaglio Controller Work

Vediamo tutte le varie annotazioni e i comportamenti che possiamo far assumere ad un controller

- **@CrossOrigin**: utilizzata per abilitare il CORS (Cross-Origin Resource Sharing) per tutti i metodi (API) di un controller. CORS è un meccanismo che consente alle risorse di essere richieste da un dominio diverso rispetto a quello in cui è ospitato il server. Senza configurare CORS, le richieste HTTP da origini diverse vengono bloccate per motivi di sicurezza.
- **@RequestMapping**: viene utilizzata per mappare le richieste HTTP a metodi del controller. Essa fornisce un modo flessibile e potente per definire come le richieste vengono instradate all'interno dell'applicazione web. In questo caso l'annotazione è a livello di classe che permette di mappare tutte le richieste che corrispondono a un certo pattern di URL a un determinato controller.

Spring offre diverse annotazioni per mappare le richieste HTTP ai metodi del Controller:

- *@GetMapping*: gestisce le richieste HTTP GET.
- *@PostMapping*: gestisce le richieste HTTP POST.
- *@PutMapping*: gestisce le richieste HTTP PUT.
- *@DeleteMapping*: gestisce le richieste HTTP DELETE.
- *@PatchMapping*: gestisce le richieste HTTP PATCH.

È possibile passare parametri o dati alla richiesta in vari modi:

- *@RequestParam*: per i parametri di query (*?param=value*).
- *@PathVariable*: per i parametri di percorso (*/entity/{id}*).
- *@RequestBody*: per il corpo della richiesta (in genere utilizzato per oggetti JSON).

Per quando riguarda la risposta, oltre a restituire semplici stringhe (message), un metodo del Controller può restituire oggetti più complessi, avvolti (wrapped) tramite un oggetto *ResponseEntity* che consente di specificare lo stato http, gli headers di risposta e chiaramente il body della risposta.

Gestione Copyright delle immagini

Due requisiti fondamentali stilati in fase di analisi sono:

- la possibilità di far visualizzare immagini protette da watermark (copyright) ad utenti registrati ed ai quali erano state fornite le autorizzazioni di visualizzazione.
- Il sistema deve garantire l'applicazione di un watermark on demand dal lato back-end, nel momento in cui viene richiesta la visualizzazione di un'immagine a risoluzione standard

In questa sezione approfondiremo la questione della protezione delle immagini con copyright, nella sezione successiva approfondiremo la questione della protezione delle API (e quindi della visualizzazione delle immagini) a livello di autenticazione e autorizzazione.

La gestione del copyright delle immagini all'interno dell'applicazione è stata progettata con l'obiettivo di prevenire l'uso non autorizzato delle risorse fotografiche e garantire la protezione dei diritti d'autore. Per fare ciò, sono state adottate diverse misure tecniche in fase di caricamento, elaborazione e visualizzazione delle immagini.

Ricordiamo, inoltre, che per un caricamento più rapido delle risorse si è deciso di sfruttare le thumbnail (immagini ridotte al 30% rispetto all'originale) che verranno utilizzate per visualizzare le griglie delle gallerie, mentre le immagini a dimensioni standard verranno richieste al server solo al momento della richiesta da parte del client.

L'amministratore del sistema ha la possibilità di caricare immagini all'interno del sistema, salvando nel database l'URL locale che punta alla posizione del NAS dove sono salvate le immagini stesse.

Per proteggere le immagini in formato ridotto, alla richiesta dell'amministratore, il sistema genera automaticamente una versione ridotta dell'immagine (thumbnail). Questo processo di

ridimensionamento riduce la dimensione dell'immagine al 30% rispetto all'originale, al fine di ridurre il consumo di banda e spazio di archiviazione.

Durante questa fase, viene anche applicato un watermark visibile all'interno delle thumbnail. Questo watermark ha la funzione di identificare l'immagine come protetta da copyright, disincentivando eventuali tentativi di utilizzo illecito. Le thumbnail così generate vengono salvate in locale sul server e sono pronte per essere inviate al client in formato MediaType png ogni volta che viene richiesta una galleria fotografica.

A differenza delle thumbnail, le immagini a dimensione standard non vengono modificate in modo permanente con l'applicazione di un watermark. Questo approccio permette di mantenere intatta la qualità e l'integrità dell'immagine originale, che potrebbe essere necessaria per usi legittimi e autorizzati.

Quando un utente richiede la visualizzazione di un'immagine nella sua dimensione reale, il watermark viene applicato dinamicamente a runtime. Questo significa che l'immagine originale non viene mai sovrascritta con il watermark, ma il sistema genera una copia temporanea con il watermark applicato, che viene poi inviata al client in formato MediaType png.

Questo meccanismo offre una doppia protezione:

- **Protezione dei diritti d'autore:** Il watermark, applicato su ogni immagine visualizzata, scoraggia l'uso non autorizzato delle immagini. Anche se l'utente tenta di scaricare l'immagine dal sito, questa sarà protetta dal watermark.
- **Conservazione dell'originale:** L'immagine originale rimane invariata e disponibile per eventuali scopi legittimi che richiedano la versione senza watermark.

Queste misure complessive garantiscono un alto livello di protezione delle immagini e dei diritti d'autore all'interno del sistema, rispettando al contempo le esigenze di usabilità e accessibilità delle risorse visive.

API security: gestione autenticazione e autorizzazione nelle API

In questa sezione si analizzerà la struttura delle API relative a Login, Registrazione e un'analisi sulla gestione dei token JWT. Questo passo ci permetterà anche di capire come sono state protette a livello di autenticazione e autorizzazione le altre API.

Prima di entrare nei dettagli dei vari componenti che configurare vediamo come funziona il flusso della procedura di registrazione di un utente e quello del login

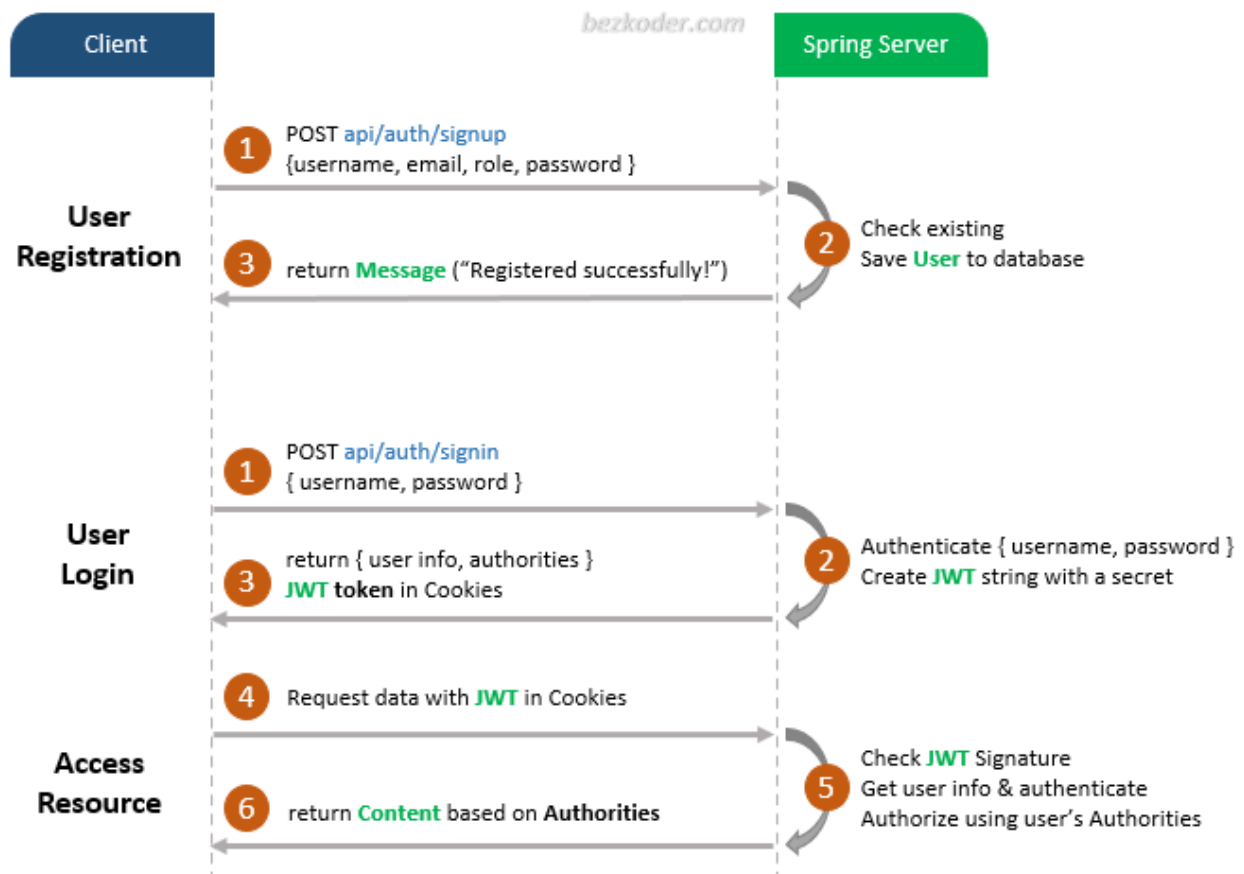


Figura 1: flusso relativo alle azioni di Registrazione, Login e accesso alle risorse

User Registration:

1. Dal client l'utente invia i propri dati (nome, cognome, username, e-mail, password)
2. Lato server viene verificata la presenza di un User già registrato sul database con i dati ricevuti, altrimenti viene persistito il nuovo User sul database e viene generato un nuovo token JWT
3. Se tutti i controlli hanno dato esito positivo, il client riceverà uno status code 200 con i dati dell'utente, il suo ruolo e il token JWT, che dovrà essere conservato e utilizzare per le richieste successive

User Login:

1. Dal client l'utente invia i propri dati per effettuare il login (e-mail, password)
2. Lato server viene verificata la correttezza dei dati ricevuti rispetto a quelli presenti sul DB
3. Se tutti i controlli hanno dato esito positivo, il client riceverà uno status code 200 con i dati dell'utente, il suo ruolo e il token JWT, che dovrà essere conservato e utilizzare per le richieste successive

Access Resource:

1. Il client effettuerà richieste API, inviando eventualmente il proprio token JWT nell'header della richiesta

2. Lato server ogni richiesta viene verificata: si controlla se quella risorsa è accessibile da tutti o dai solo soggetti in possesso di token JWT valido e si verifica che l'utente che ha fatto la richiesta abbia il ruolo per poterla fare
3. Se tutti i controlli hanno dato esito positivo, il client riceverà uno status code 200 con i dati. Altrimenti riceverà uno status code 400 o 403.

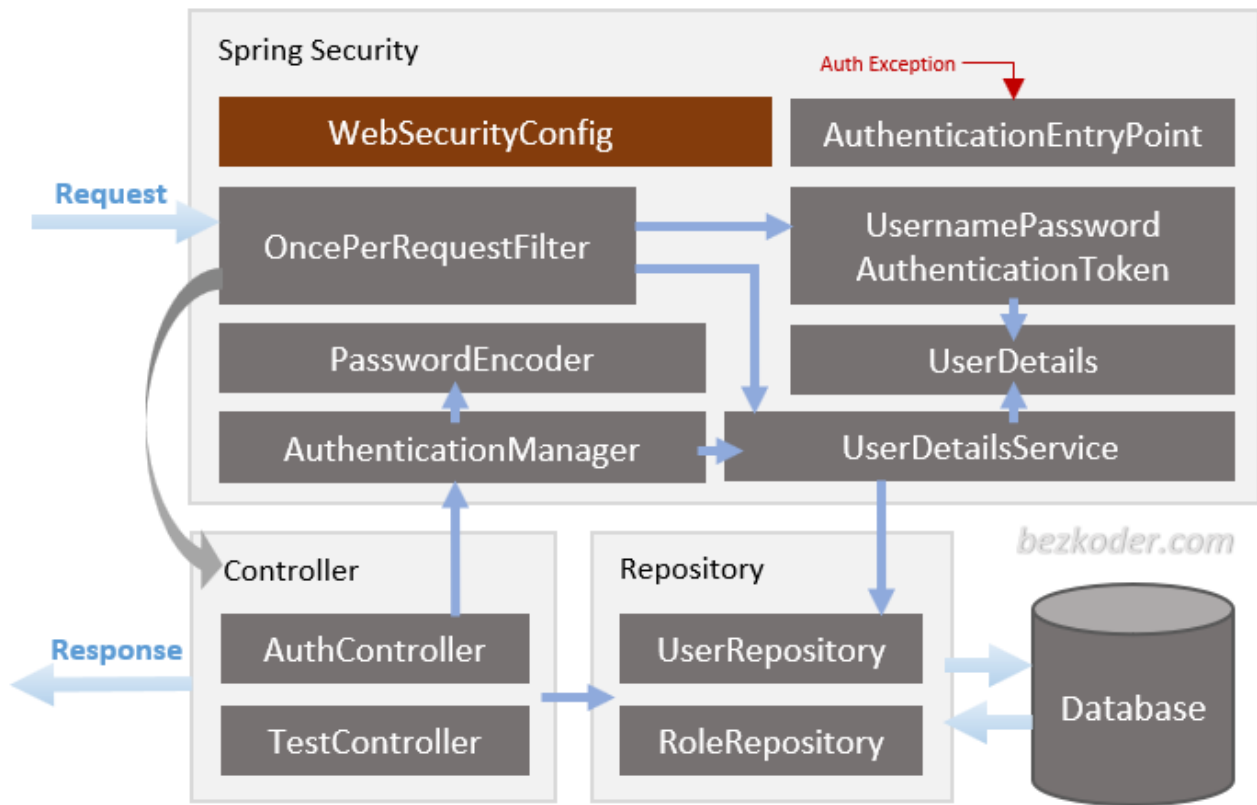


Figura 2: Schema Request->Response del framework Spring Security

Nell'immagine precedente possiamo vedere ad alto livello tutti i componenti necessari per proteggere (a livello di autenticazione e autorizzazione) le risorse API:

- **WebSecurityConfig:** è il componente principale di tutta l'architettura di sicurezza. Configura cors, csrf, gestione delle sessioni, regole per le risorse protette, garantisce e performa authentication e authorization.
- **UserDetailsService:** l'interfaccia ha un metodo per caricare l'utente in base al nome utente e restituisce un oggetto UserDetails che Spring Security può utilizzare per l'autenticazione e la convalida.
- **UserDetails:** contiene le informazioni necessarie (ad esempio: nome utente, password, role) per creare un oggetto usato nel processo di autenticazione.
- **UsernamePasswordAuthenticationToken:** è una classe utilizzata per rappresentare un token di autenticazione basato su nome utente e password. Questo token viene comunemente usato durante il processo di autenticazione di un utente. Il token viene poi passato al gestore di autenticazione (AuthenticationManager) per verificare la validità delle credenziali.

- **AuthenticationManager:** è un'interfaccia che rappresenta il componente centrale responsabile della gestione del processo di autenticazione. È utilizzato per verificare le credenziali degli utenti e determinare se un utente è autenticato con successo o meno.
- **OncePerRequestFilter:** è progettato per essere utilizzato per implementare filtri che devono essere eseguiti una sola volta per richiesta API. Viene utilizzato per analizzare i token JWT (JSON Web Token) per autenticare le richieste.
- **AuthenticationEntryPoint:** Quando un utente tenta di accedere a una risorsa protetta senza essere autenticato, l'AuthenticationEntryPoint viene invocato per gestire questo scenario. In generale, il suo compito è indirizzare l'utente verso una pagina di login o restituire una risposta HTTP adeguata (ad esempio, un errore 401 Unauthorized).

Vediamo adesso più nel dettaglio (con approccio bottom-up, dal modulo di configurazione all'endpoint che accoglie le richieste) tutte le singole classi realizzate per l'autenticazione degli utenti e della gestione dell'autorizzazione delle risorse:

- it.myportfolio.security -> WebSecurityConfig
- it.myportfolio.service -> UserDetailsServiceImpl implements UserDetailsService
- it.myportfolio.dto -> UserDetailsImpl implements UserDetails
- it.myportfolio.security.jwt -> AuthEntryPointJwt implements AuthenticationEntryPoint
- it.myportfolio.security.jwt -> AuthTokenFilter extends OncePerRequestFilter
- it.myportfolio.security.jwt -> JwtUtils
- it.myportfolio.controller -> AuthController
- it.myportfolio.repository -> UserRepository extends JpaRepository<User, Long> (già analizzato in precedenza)
- it.myportfolio.repository -> RoleRepository extends JpaRepository<Role, Long> (già analizzato in precedenza)
- it.myportfolio.model -> User (già analizzato in precedenza)
- it.myportfolio.model -> Role: id, name (già analizzato in precedenza)
- it.myportfolio.security.payload contiene le classi DTO per interfacciarsi con i client nelle operazioni di richiesta e risposta al login/registrazione

Per lavorare con il modulo Spring Security è necessario aggiungere le seguenti dipendenze del pom.xml:

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.11.5</version>
</dependency>

<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-impl</artifactId>
  <version>0.11.5</version>
  <scope>runtime</scope>
</dependency>

<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId>
  <version>0.11.5</version>
  <scope>runtime</scope>
</dependency>

```

Snippet 23: dipendenze necessarie per lavorare con il framework Spring Security

WebSecurityConfig

L'obiettivo principale di questa classe è definire la sicurezza a livello di applicazione, configurando il flusso di autenticazione e autorizzazione.

Vediamo in dettaglio le parti principali di questa classe:

Annotazioni

- **@Configuration:** Indica che questa classe è una classe di configurazione Spring. Consente di definire dei bean che saranno gestiti dal container di Spring.
- **@EnableMethodSecurity:** Abilita i vari moduli Spring relativi alla security. Questa annotazione consente di applicare regole di sicurezza a livello di metodo utilizzando annotazioni come @PreAuthorize.

Dipendenze Autowired

- **@Autowired UserDetailsServiceImpl:** viene utilizzata per caricare i dettagli di un utente durante l'autenticazione, tipicamente recuperandoli da un database.
- **@Autowired private AuthEntryPointJwt:** componente AuthenticationEntryPoint, che gestisce le richieste non autorizzate. Quando una richiesta non autenticata tenta di accedere a una risorsa protetta, l'oggetto unauthorizedHandler gestirà la risposta (restituendo un errore 401 Unauthorized).

Bean

SecurityFilterChain filterChain(HttpSecurity http): questo metodo configura la catena di filtri di sicurezza (*SecurityFilterChain*) utilizzando l'API di *HttpSecurity*. Questo metodo viene chiamato per analizzare tutte le richieste che arrivano al web server. Vediamo questo metodo passo-passo:

- **Disabilitazione CSRF:** *http.csrf(csrf -> csrf.disable())*
 - CSRF (Cross-Site Request Forgery) viene disabilitato. È comune in applicazioni RESTful, dove l'autenticazione si basa su token (come JWT) anziché su sessioni di login.
- **Gestione delle Eccezioni:** *exceptionHandling(exception -> exception.authenticationEntryPoint(unauthorizedHandler))*
 - viene configurato il gestore delle eccezioni, utilizzando *unauthorizedHandler* come *AuthenticationEntryPoint*. Questo componente sarà invocato quando una richiesta non autenticata tenta di accedere a una risorsa protetta.
- **Session Management:** *sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))*
 - Viene configurata la gestione delle sessioni come **stateless**, il che significa che l'applicazione non manterrà lo stato delle sessioni tra le richieste.
- **Autorizzazione delle Richieste:** *authorizeHttpRequests(auth -> auth...*
 - Vengono definite le rotte (*/api/auth/***, */api/user/***, ecc.) pubbliche.
- **Configurazione del Provider di Autenticazione:**
http.authenticationProvider(authenticationProvider())
 - Viene registrato il provider di autenticazione configurato in precedenza (*DaoAuthenticationProvider*) come provider di autenticazione da utilizzare.
- **Aggiunta del Filtro JWT:** *http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class)*
 - Viene aggiunto il filtro JWT alla catena dei filtri di sicurezza, prima del filtro *UsernamePasswordAuthenticationFilter*. Questo significa che il filtro JWT verrà eseguito prima della gestione dell'autenticazione basata su nome utente e password.

```

@Configuration
@EnableMethodSecurity
public class WebSecurityConfig {

    @Autowired
    UserDetailsServiceImpl userDetailsService;

    @Autowired
    private AuthEntryPointJwt unauthorizedHandler;

    @Bean
    protected AuthTokenFilter authenticationJwtTokenFilter() {
        return new AuthTokenFilter();
    }

    @Bean
    protected DaoAuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authProvider = new
        DaoAuthenticationProvider();

        authProvider.setUserDetailsService(userDetailsService);
        authProvider.setPasswordEncoder(passwordEncoder());

        return authProvider;
    }

    @Bean
    protected AuthenticationManager
authenticationManager(AuthenticationConfiguration authConfig) throws Exception {
        return authConfig.getAuthenticationManager();
    }

    @Bean
    protected PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    protected SecurityFilterChain filterChain(HttpSecurity http) throws
Exception {
        http.csrf(csrf -> csrf.disable())
            .exceptionHandling(exception ->
exception.authenticationEntryPoint(unauthorizedHandler))
            .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(auth ->
auth.requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/api/user/**").permitAll()
                .requestMatchers("/api/work/**").permitAll()
                .requestMatchers("/api/image/**").permitAll()
                .requestMatchers("/api/cart/**").permitAll()
                .requestMatchers("/api/shop/**").permitAll()

.requestMatchers("/api/shopableimage/**").permitAll()
                .anyRequest().authenticated());

        http.authenticationProvider(authenticationProvider());
        http.addFilterBefore(authenticationJwtTokenFilter(),
UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }
}

```


UserDetailsServiceImpl

Layer service che permette di recuperare le informazioni di un utente fornendo in input un username. Viene controllato anche con l'utente sia abilitato, altrimenti solleva un'eccezione.

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUsernameAndEnable(username, true)
            .orElseThrow(() -> new UsernameNotFoundException("User Not Found or
disable with username: " + username));

        return UserDetailsImpl.build(user);
    }
}
```

UserDetailsImpl

La classe UserDetailsImpl implementa l'interfaccia UserDetails di Spring Security, che rappresenta gli attributi di un utente che verranno utilizzati per l'autenticazione e l'autorizzazione.

Dettaglio da notare è *Collection<? extends GrantedAuthority> authorities*: una collezione di autorizzazioni (GrantedAuthority) che rappresentano i ruoli o le autorizzazioni assegnati all'utente e sono utilizzate per gestire l'autorizzazione all'interno dell'applicazione.

Implementando UserDetails, si è obbligati a implementare i metodi definiti da questa interfaccia, che sono utilizzati da Spring Security per autenticare e autorizzare gli utenti. Anche se questi metodi non sono visibili nel codice fornito, sono:

1. **getAuthorities()**: Restituisce la collezione di GrantedAuthority (ruoli/permessi) dell'utente.
2. **getPassword()**: Restituisce la password dell'utente.
3. **getUsername()**: Restituisce il nome utente dell'utente.

```

public class UserDetailsImpl implements UserDetails {
    private static final long serialVersionUID = 1L;

    private Long id;

    private String username;

    private String email;

    @JsonIgnore
    private String password;

    private Collection<? extends GrantedAuthority> authorities;

    public UserDetailsImpl(Long id, String username, String email, String
password,
        Collection<? extends GrantedAuthority> authorities) {
        this.id = id;
        this.username = username;
        this.email = email;
        this.password = password;
        this.authorities = authorities;
    }

    public static UserDetailsImpl build(User user) {
        List<GrantedAuthority> authorities = user.getRoles().stream()
            .map(role -> new
SimpleGrantedAuthority(role.getName().name()))
            .collect(Collectors.toList());

        return new UserDetailsImpl(user.getId(), user.getUsername(),
user.getEmail(), user.getPassword(), authorities);
    }

    ...
    getters and setters methods
    ...
}

```

Snippet 26: Dettaglio DTO *UserDetailsImpl*

AuthTokenFilter

La classe *AuthTokenFilter* estende la classe *OncePerRequestFilter*, progettata per garantire che il filtro venga eseguito una sola volta per ogni richiesta HTTP. Questo filtro viene utilizzato per gestire l'autenticazione basata su JWT (JSON Web Token).

Il filtro verifica la presenza all'interno della richiesta di un token valido. Si costruisce il modello dell'utente autenticato, recuperando i valori dal database.

Viene settato il *SecurityContextHolder.getContext().setAuthentication(authentication)*: è il meccanismo di Spring Security per mantenere il contesto di sicurezza per il thread corrente. In altre parole, Spring Security ora riconosce l'utente come autenticato per il resto della richiesta.

Infine la richiesta viene passata al filtro successivo nella catena di filtri, indipendentemente dal fatto che l'autenticazione sia avvenuta o meno. Questo è importante per garantire che la richiesta possa proseguire attraverso altri filtri e componenti del sistema.

```

public class AuthTokenFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtils jwtUtils;

    @Autowired
    private UserDetailsServiceImpl userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
    HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
        try {
            String jwt = jwtUtils.getJwtFromCookies(request);
            if (jwt != null && jwtUtils.validateJwtToken(jwt)) {
                String username = jwtUtils.getUserNameFromJwtToken(jwt);

                UserDetails userDetails =
                userDetailsService.loadUserByUsername(username);

                UsernamePasswordAuthenticationToken authentication = new
                UsernamePasswordAuthenticationToken(
                    userDetails, null, userDetails.getAuthorities());

                authentication.setDetails(new
                WebAuthenticationDetailsSource().buildDetails(request));

                SecurityContextHolder.getContext().setAuthentication(authentication);
            } catch (Exception e) {

            }

            filterChain.doFilter(request, response);
        }
    }
}

```

Snippet 27: configurazione filtro AuthTokenFilter

AuthEntryPointJwt

La classe AuthEntryPointJwt implementa l'interfaccia AuthenticationEntryPoint di Spring Security, che viene utilizzata per gestire le richieste non autenticate o per rispondere quando un utente tenta di accedere a una risorsa protetta senza essere autenticato.

Il metodo *commence()* gestisce la risposta HTTP quando si verifica un errore di autenticazione risponde con uno status code HTTP 401 (Unauthorized) indicando che l'utente non è autorizzato.

```

@Component
public class AuthEntryPointJwt implements AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
        AuthenticationException authException)
        throws IOException, ServletException {
        response.setContentType(MediaType.APPLICATION_JSON_VALUE);
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);

        final Map<String, Object> body = new HashMap<>();
        body.put("status", HttpServletResponse.SC_UNAUTHORIZED);
        body.put("error", "Unauthorized");
        body.put("message", authException.getMessage());
        body.put("path", request.getServletPath());

        final ObjectMapper mapper = new ObjectMapper();
        mapper.writeValue(response.getOutputStream(), body);
    }
}

```

Snippet 28: configurazione filtro AuthEntryPointJwt

AuthController

Questo controller è il punto di ingresso per le richieste da parte dei client per login, registrazione e logout.

```

@PostMapping("/signin")
public ResponseEntity<?> authenticateUser(@RequestBody LoginRequest loginRequest) {

    Authentication authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(loginRequest.getUsername(),
        loginRequest.getPassword()));

    SecurityContextHolder.getContext().setAuthentication(authentication);

    UserDetailsImpl userDetails = (UserDetailsImpl) authentication.getPrincipal();

    ResponseCookie jwtCookie = jwtUtils.generateJwtCookie(userDetails);

    List<String> roles = userDetails.getAuthorities().stream().map(item -> item.getAuthority())
        .collect(Collectors.toList());

    return ResponseEntity.ok().header(HttpHeaders.SET_COOKIE, jwtCookie.toString()).body(
        new UserInfosResponse(userDetails.getId(), userDetails.getUsername(),
        userDetails.getEmail(), roles));
}

@PostMapping("/signup")
public ResponseEntity<?> registerUser(@RequestBody SignupRequest signUpRequest) {

    if (userRepository.existsByUsername(signUpRequest.getUsername())) {
        return ResponseEntity.badRequest().body(new MessageResponse("Error: Username is already
        taken!"));
    }

    if (userRepository.existsByEmail(signUpRequest.getEmail())) {
        return ResponseEntity.badRequest().body(new MessageResponse("Error: Email is already in
        use!"));
    }

    Set<Role> roles = new HashSet<>();
    User user = new User(signUpRequest.getUsername(), signUpRequest.getEmail(),
        encoder.encode(signUpRequest.getPassword()), signUpRequest.getSurname(),
        signUpRequest.getName());
    Role userRole = roleRepository.findByName(ERole.ROLE_USER)
        .orElseThrow(() -> new RuntimeException("Error: Role is not found."));
    roles.add(userRole);

    user.setRoles(roles);
    user.setEnabled(true);
    userRepository.save(user);

    return ResponseEntity.ok(new MessageResponse("User registered successfully!"));
}

@PostMapping("/signout")
public ResponseEntity<?> logoutUser() {
    ResponseCookie cookie = jwtUtils.getCleanJwtCookie();
    return ResponseEntity.ok().header(HttpHeaders.SET_COOKIE, cookie.toString())
        .body(new MessageResponse("You've been signed out!"));
}

```

Snippet 29: Dettaglio Controller AuthController

JwtUtils

La classe JwtUtils è una classe di utilità per la gestione dei token JWT (JSON Web Token). Questa classe fornisce metodi per creare, validare e estrarre informazioni dai token JWT, e per gestire i cookie che contengono tali token.

Questa classe è fondamentale per un'architettura basata su JWT in cui l'autenticazione e l'autorizzazione sono gestite in modo stateless (senza sessione) attraverso token che vengono scambiati tra client e server.

Le variabili di classe utili per la gestione dei token sono:

- **jwtSecret:** Una stringa segreta utilizzata per firmare i token JWT. Questa chiave è crittografata utilizzando l'algoritmo HMAC SHA.
- **jwtExpirationMs:** Indica la durata in millisecondi della validità del token JWT (in questo caso 24 ore).
- **jwtCookie:** Nome del cookie che contiene il token JWT (impostato a "Bearer").

Metodi Principali

- **getJwtFromCookies(HttpServletRequest request):** questo metodo estrae il token JWT dai cookie presenti nella richiesta HTTP.
- **generateJwtCookie(UserDetailsImpl userPrincipal):** utilizza il metodo `generateTokenFromUsername()` per creare un token JWT basato sul nome utente e sull'ID dell'utente. Poi, costruisce un cookie HTTP-only (contenente il token JWT, con una durata massima di 24 ore).
- **getCleanJwtCookie():** questo metodo può essere utilizzato per invalidare un cookie JWT fornito in input.
- **getUserNameFromJwtToken(String token):** utilizza la libreria JWT (io.jsonwebtoken) per analizzare il token JWT, decodificandolo con la chiave segreta e restituendo lo username memorizzato nel corpo del token.
- **getUserIdFromJwtToken(String token):** questo metodo analizza il token JWT e restituisce l'ID utente che è memorizzato.
- **key():** decodifica la stringa `jwtSecret` in un array di byte utilizzando `Decoders.BASE64.decode()` e genera una chiave HMAC SHA (`hmacShaKeyFor`) che verrà utilizzata per firmare o verificare i token.
- **validateJwtToken(String authToken):** prova a eseguire il parsing del token JWT. Se l'operazione ha successo, il token è valido e viene restituito `true`. Se si verifica un'eccezione, come un token scaduto, malformato o non supportato, viene restituito `false`.
- **generateTokenFromUsername(String username, Long id):** costruisce un token JWT impostando il subject (nome utente), aggiungendo l'ID dell'utente come claim personalizzato, impostando la data di emissione e la data di scadenza del token, e infine firmando il token con l'algoritmo HMAC SHA256 e la chiave generata dal metodo `key()`.

API security: gestione sicurezza immagini

Come visto in precedenza la visualizzazione dei Work è protetta in dalla visualizzazione non autorizzata a livello di query. Le singole immagini sono protette dal watermark, ma un'utente potrebbe tentare l'accesso alle immagini a dimensione intera facendo richiesta all'API sotto, provando una serie di id.

Per evitare che visualizzi immagini appartenenti a Work per i quali non ha accesso, viene recuperato dal DB il Work a cui appartiene l'immagine e viene verificato che l'utente ne abbia

effettivamente diritto di visualizzazione. In caso positivo viene restituita l'immagine protetta dal watermark, in caso contrario viene restituito uno status code FORBIDDEN.

```
@GetMapping()
@PreAuthorize("hasRole('USER') or hasRole('ADMIN')")
public ResponseEntity<byte[]> getImageById(@RequestParam Long id,
                                           HttpServletRequest request) throws IOException {

    ImageProject image = imageService.getImageById(id);
    if (image != null) {
        String token = jwtUtils.getJwtFromCookies(request);
        if (jwtUtils.validateJwtToken(token)) {
            Long userId = (Long) jwtUtils.getUserIdFromJwtToken(token);
            Optional<User> user = userService.getUserById(userId);
            Work work = workService.getWorkById(userId);

            if (work.getUsers().contains(user.get())) {
                BufferedImage sourceImage = ImageIO.read(new
                    File(image.getUrl()));

                // Add the watermark using the addTextWatermark
                // function
                BufferedImage watermarkedImage =
                    ThumbnailGenerator.addTextWatermark(sourceImage);

                // Converti BufferedImage in array di byte
                byte[] imageBytes =
                    convertImageToBytes(watermarkedImage);

                return ResponseEntity.status(HttpStatus.OK)
                    .header(org.springframework.http.HttpHeaders.CONTENT_DISPOSITION)
                    .contentType(MediaType.IMAGE_PNG).body(imageBytes);
            } else {
                return ResponseEntity.status(HttpStatus.FORBIDDEN)
                    .build();
            }
        } else {
            return ResponseEntity.status(
                HttpStatus.UNAUTHORIZED).build();
        }
    }
    return ResponseEntity.notFound().build();
}
```

Snippet 30: dettaglio delle risorse Image protette da Role e da visualizzazione non lecita

Descrizione componenti di utility

Oltre ai componenti visti in precedenza che rappresentano i pilastri del progetto sono state realizzate anche le due classi di “servizio” *BlockchainTransactionService* e *ThumbnailGenerator*, che forniscono metodi ausiliari per la registrazione di nuovi blocchi all'interno della Blockchain e l'applicazione di watermark sulle immagini.

La classe *ThumbnailGenerator* assolve due compiti:

- La generazione Thumbnail a partire da un oggetto ImageProject. Recupera dai parametri in input il path in cui è contenuta l'immagine originale e il path dove andare a salvare le Thumbnail generate con la classe Image del package java.awt.Image. La classe permette di ridurre la dimensione al 30% dell'immagine originale.
- L'applicazione del watermark "© MyPortfolio 2024" in rosso, con il testo semi-trasparente con un'opacità dell'80% e nella parte bassa di un'immagine di tipo BufferedImage fornita in input

```
public static void makeThumbnail(ImageProject image) throws IOException {
    // controlli e/o creazione path di destinazione
    ...

    BufferedImage originalImage = ImageIO.read(new File(photoUrl));

    int lengthThumbnail = (int) (originalImage.getWidth() * 0.3);
    int widthThumbnail = (int) (originalImage.getHeight() * 0.3);

    BufferedImage thumbnail = new BufferedImage(lengthThumbnail, widthThumbnail,
        BufferedImage.TYPE_INT_RGB);

    Graphics2D g2 = thumbnail.createGraphics();
    g2.drawImage(originalImage, 0, 0, lengthThumbnail, widthThumbnail, null);
    g2.drawImage(originalImage.getScaledInstance(widthThumbnail, lengthThumbnail,
        Image.SCALE_SMOOTH), 0, 0, null);
    g2.dispose();

    // Add watermark to the thumbnail
    thumbnail = addTextWatermark(thumbnail);

    if (thumbnail != null) {
        ImageIO.write(thumbnail, ext, new File(thumbnailPath.toString()));
    } else {
        System.err.println("Failed to add watermark to the thumbnail.");
    }
}
```

Snippet 31: funziona makeThumbnail che si occupa di generare e salvare nel path indicato, una nuova immagine ridotta del 30% rispetto all'originale. La parte di verifica della presenza dell'immagine, la verifica e/o creazione del path di destinazione dell'immagine creata è omessa.


```

public static BufferedImage addTextWatermark(BufferedImage sourceImage) {
    try {
        String text = "© MyPortfolio 2024";
        Graphics2D g2d = (Graphics2D) sourceImage.getGraphics();

        // initializes necessary graphic properties
        AlphaComposite alphaChannel = AlphaComposite.getInstance
            (AlphaComposite.SRC_OVER, 0.8F);

        g2d.setComposite(alphaChannel);
        g2d.setColor(Color.RED);
        g2d.setFont(new Font("Arial", Font.BOLD, 32));
        FontMetrics fontMetrics = g2d.getFontMetrics();
        Rectangle2D rect = fontMetrics.getStringBounds(text, g2d);

        // calculates the coordinate where the String is painted
        int centerX = (sourceImage.getWidth() -
            (int) rect.getWidth()) / 2;
        int centerY = (int) ((int) sourceImage.getHeight() / 1.1F);

        // paints the textual watermark
        g2d.drawString(text, centerX, centerY);

        g2d.dispose();

        return sourceImage;
    } catch (Exception ex) {
        System.err.println(ex);
        return null;
    }
}

```

Snippet 32: funzione che presa in input una Buffered Image restituisce un nuovo oggetto BufferedImage, ma con l'aggiunta di una filigrana testuale (con testo semi-trasparente con un'opacità dell'80%.) applicata nella parte bassa dell'immagine e centrato rispetto all'ascisse.

La classe *BlockchainTransactionService* assolve due compiti:

- Registrazione di un nuovo acquisto come nuovo blocco all'interno della Blockchain
- Verifica della veridicità di un hash di un blocco

Vista l'importanza della parte dedicata alla Blockchain, per non rischiare i fare troppa sintesi di questi aspetti, tutte le componenti in gioco verranno riprese e espanse nella sezione “Sezione shop e blockchain”

Implementazione lato DB

I tool utilizzati per lo strato di persistenza sono:

- ORM: Hibernate 6.4.4 Final
- RDBMS: MariaDB 11.2.
- Tool di amministrazione: HeidiSQL 12.3.0.6589

In questa sezione analizziamo nel dettaglio come il framework Hibernate ha gestito (sia con le annotazioni inserite nelle classi dei Model, sia con i propri automatismi) la persistenza dei dati sul database dei modelli definiti come oggetti Java.

Le tabelle create sul database, a partire dai model sono le seguenti:

Nome ^	Motore	Tipo
cart	InnoDB	Table
cart_shopable_image	InnoDB	Table
hibernate_sequences	InnoDB	Table
image	InnoDB	Table
roles	InnoDB	Table
sales_order	InnoDB	Table
shopable_image	InnoDB	Table
user	InnoDB	Table
user_roles	InnoDB	Table
visible_work	InnoDB	Table
work	InnoDB	Table

Segue adesso un dettaglio delle singole tabelle, riportate nello stesso ordine dell'immagine precedente, con alcuni commenti nelle tabelle più interessanti

cart

#	Nome	Tipo di dati	Lunghezza/set	Senza segno	Permetti NULL	Riem...	Predefinito
1	id	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	AUTO_INCREMENT
2	user_id	BIGINT	20	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL

cart_shopable_image

Questa tabella permette di registrare quali sono le ShopableImage che un certo utente ha deciso di aggiungere al proprio carrello, per poi eventualmente procedere con l'acquisto.

#	Nome	Tipo di dati	Lunghezza/set	Senza segno	Permetti NULL	Riempì con zero	Predefinito
1	cart_id	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Nessun valore predefinito
2	shopable_image_id	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Nessun valore predefinito

hibernate_sequences

La tabella hibernate_sequences viene creata automaticamente da Hibernate quando si utilizza la strategia di generazione delle chiavi primarie GenerationType.TABLE, che permette di generare identificatori unici per le entità gestite da Hibernate.

Quando si decide di utilizzare GenerationType.TABLE viene utilizzata una tabella specifica per tenere traccia dei numeri di sequenza utilizzati per generare gli ID delle entità.

La tabella hibernate_sequences contiene due colonne:

1. **sequence_name**: il nome della sequenza (o tabella) per la quale si stanno generando gli ID.
2. **next_val**: il valore successivo che verrà utilizzato per l'ID.

Quando Hibernate genera un nuovo ID per un'entità, consulta questa tabella per ottenere il valore corrente, incrementa il valore e aggiorna la tabella con il nuovo valore. Questo meccanismo garantisce che ogni entità abbia un ID univoco.

```
Hibernate: create table hibernate_sequences (next_val bigint, sequence_name
varchar(255) not null, primary key (sequence_name)) engine=InnoDB
```

#	Nome	Tipo di dati	Lunghezza/set	Senza segno	Permetti NULL	Riempi con zero	Predefinito
1	next_val	BIGINT	20	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
2	sequence_name	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Nessun valore predefinito

image

Su questa tabella vengono salvate le entità create tramite il model ImageProject, il parametro work_id specifica a quale Work deve essere associata l'immagine.

Per definizione di Image all'interno dei requisiti, tutti i parametri in questa tabella non possono essere NULL.

#	Nome	Tipo di dati	Lunghezza/set	Senza segno	Permetti NULL	Riempi con zero	Predefinito
1	id	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Nessun valore predefinito
2	work_id	BIGINT	20	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
3	label	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
4	thumbnailurl	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
5	url	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL

roles

#	Nome	Tipo di dati	Lunghezza/set	Senza segno	Permetti NULL	Riempi con zero	Predefinito
1	id	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	AUTO_INCREMENT
2	name	ENUM	'ROLE_USER', '...	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL

sales_order

Questa tabella permette di registrare gli acquisti fatti dagli utenti; oltre all'ID dell'utente che ha fatto l'acquisto, viene salvato il timestamp e l'hash (o digest) che viene restituito dallo script di registrazione nuovo block dettagliato nella sezione "Sezione shop e blockchain"

#	Nome	Tipo di dati	Lunghezza/set	Senza segno	Permetti NULL	Riempi con zero	Predefinito
1	id	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	AUTO_INCREMENT
2	timestamp	DATETIME	6	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
3	user_id	BIGINT	20	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
4	hash	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL

shopable_image

Questa tabella memorizza le immagini che l'admin ha deciso di mettere in vendita all'interno dello shop. Poiché la classe (model) ShopableImage è realizzata sfruttando l'ereditarietà a partire dal model ImageProject e poiché è stata utilizzata l'annotazione @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS), allora all'interno del database ogni classe concreta nella gerarchia di ereditarietà ha la propria tabella separata nel database.

Ogni tabella contiene tutte le colonne necessarie per rappresentare completamente le istanze delle rispettive sottoclassi.

Oltre ai campi ereditati, questa classe ha in più i seguenti parametri:

- **price:** che indica il prezzo di vendita
- **sales_order_id:** se compilato indica l'id dell'ordine con il quale è stata acquistata l'immagine
- **is_sold:** indica se quell'immagine è stata acquistata o meno

Quando viene aggiunta una nuova immagine alla vendita, questa farà parte del Work con ID 0.

#	Nome	Tipo di dati	Lunghezza/set	Senza segno	Permetti NULL	Riempi con zero	Predefinito
1	is_sold	TINYINT	1	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	'0'
2	price	FLOAT		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Nessun valore predefinito
3	id	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Nessun valore predefinito
4	sales_order_id	BIGINT	20	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
5	work_id	BIGINT	20	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
6	label	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
7	thumbnailurl	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
8	url	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Nessun valore predefinito

user

#	Nome	Tipo di dati	Lunghezza/set	Senza segno	Permetti NULL	Riempi con zero	Predefinito
1	enable	BIT	1	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
2	id	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	AUTO_INCREMENT
3	email	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
4	name	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
5	password	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
6	surname	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
7	username	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL

user_roles

#	Nome	Tipo di dati	Lunghezza/set	Senza segno	Permetti NULL	Riempi con zero	Predefinito
1	role_id	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Nessun valore predefinito
2	user_id	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Nessun valore predefinito

visible_work

#	Nome	Tipo di dati	Lunghezza/set	Senza segno	Permetti NULL	Riempi con zero	Predefinito
1	user_id	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Nessun valore predefinito
2	work_id	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Nessun valore predefinito

work

#	Nome	Tipo di dati	Lunghezza/set	Senza segno	Permetti NULL	Riempi con zero	Predefinito
1	completion_date	DATETIME	6	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
2	id	BIGINT	20	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	AUTO_INCREMENT
3	company	VARCHAR	255	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	NULL
4	title	VARCHAR	255	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	Nessun valore predefinito

Implementazione front-end

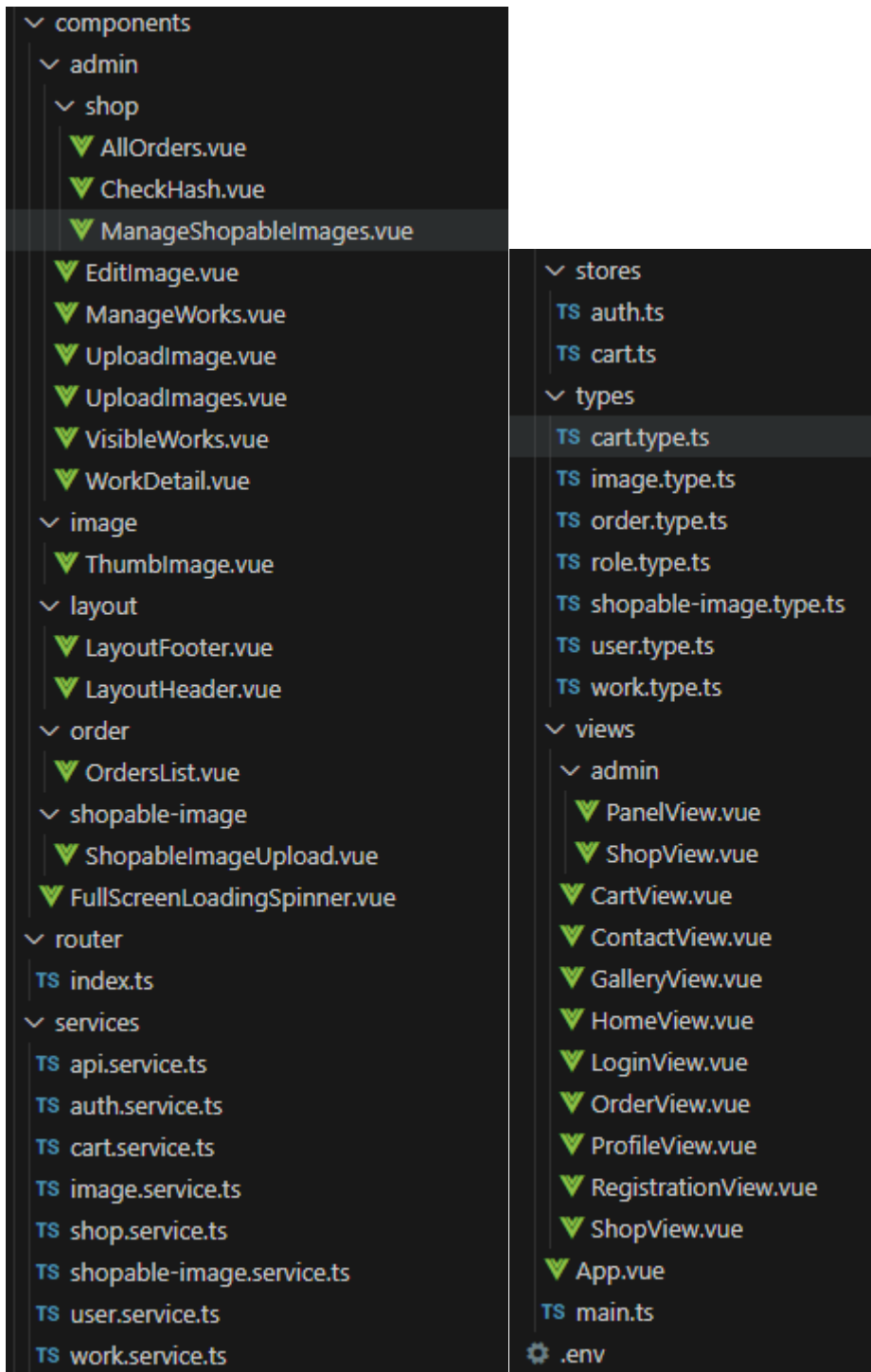
Il front-end dell'applicazione è stato sviluppato utilizzando HTML e Vue.js, con l'obiettivo di creare un'interfaccia utente intuitiva, dinamica e reattiva. L'intento principale è quello di offrire agli utenti un'esperienza fluida e coinvolgente nella gestione e nella visualizzazione dei lavori fotografici, garantendo al contempo facilità d'uso e un design moderno.

L'utilizzo di Vue.js come framework front-end ha permesso di sfruttare la sua architettura basata su componenti, che facilita la modularità del codice e semplifica l'implementazione di funzionalità interattive.

In questa sezione saranno descritti i principali aspetti relativi allo sviluppo del front-end, includendo:

- La struttura di alcuni componenti Vue.js e il loro ruolo nel progetto;
- Il test di alcune funzionalità

Struttura del front-end



Il codice front-end è organizzato in diversi moduli e cartelle per mantenere un'architettura chiara e modulare. Le principali componenti e file sono suddivisi come segue:

- **components:** contiene i componenti Vue riutilizzabili all'interno dell'applicazione. Ad esempio, nella sottocartella shop, troviamo componenti come ManageShopableImages.vue e UploadImages.vue, che gestiscono specifiche funzionalità del portfolio, come l'upload di immagini o la gestione delle immagini acquistabili all'interno dello shop. I componenti LayoutHeader.vue e LayoutFooter.vue sono dedicati alla struttura generale dell'applicazione, in particolare il menù ed il footer dell'applicazione.
- **services:** contiene file .ts (TypeScript) che implementano servizi centralizzati per comunicare con le API. Ad esempio, auth.service.ts gestisce l'autenticazione, mentre image.service.ts si occupa delle operazioni relative alle immagini, come il caricamento, l'eliminazione e l'aggiornamento.
- **types:** raccoglie i file TypeScript che definiscono i tipi e le interfacce utilizzate nel progetto per mantenere un typing rigoroso. Ad esempio, image.type.ts e user.type.ts definiscono rispettivamente i tipi per le immagini e gli utenti.
- **views:** contiene le pagine principali dell'applicazione, ciascuna rappresentata da un file Vue. Ad esempio, HomeView.vue è la vista principale, mentre LoginView.vue e RegistrationView.vue gestiscono rispettivamente le funzionalità di accesso e registrazione degli utenti. La suddivisione in sottocartelle (admin, ecc.) organizza le viste per user Role.
- **stores:** implementa la gestione dello stato centralizzata, sfruttando strumenti come Vuex e Pinia. File come auth.ts e cart.ts gestiscono lo stato legato all'autenticazione e al carrello.
- **File principali:** App.vue è il punto di ingresso principale del progetto, che definisce la struttura di base dell'applicazione, mentre main.ts inizializza il progetto Vue e configura librerie come il router o lo store.
- **Router:** gestisce la navigazione tra le diverse pagine (o viste) dell'applicazione e include dei controlli di navigazione per autorizzare l'accesso alle rotte in base allo stato di autenticazione e ai ruoli dell'utente.

Vediamo nel dettaglio alcuni esempi di oggetti per capire un po' più nel dettaglio le loro funzionalità

AllOrders.vue

```
<script lang="ts" setup>
import OrdersList from '@components/order/OrdersList.vue'
import { getAllOrders } from '@services/shop.service'
import type { Order } from '@types/order.type'
import { onMounted, ref } from 'vue'

const orders = ref<Order[]>([])

const loadOrders = async () => {
  orders.value = await getAllOrders()
}
onMounted(() => {
  loadOrders()
})
</script>
<template>
  <div class="mt-4">
    <div v-if="orders.length > 0">
      <h2 class="mb-5">Ordini</h2>
      <OrdersList :orders="orders" :show-hash="true" />
    </div>
    <div v-else class="d-flex align-items-center flex-column pt-5">
      <i class="bi bi-shop h1"></i>
      <h3>Nessun ordine trovato</h3>
    </div>
  </div>
</template>
```

Questo component sfruttando il service `getAllOrders`, che recupera gli ordini dal DB, permette all'admin di visualizzare in dei riquadri tutti gli ordini fatti all'interno dello shop. In particolare la funzione `onMounted` crea la pagina al "montaggio" dell'applicazione (appena viene caricata la pagina).

shop.service.ts

```
import type { Order, OrderDetail } from '@types/order.type'
import { get, post } from './api.service'

export const makeOrder = async () => {
  const response = await post('api/shop')
  if (!response.ok) {
    const body = await response.json()
    throw new Error(body.message ?? 'Si è verificato un errore durante la creazione dell\'ordine')
  }
}

export const getMyOrders = async (): Promise<Order[]> => {
  const response = await get('api/shop/myorder')
  const body = await response.json()
  if (response.ok) {
    return body
  } else {
    throw new Error(body.message ?? 'Si è verificato un errore durante il recupero degli ordini')
  }
}
...
```

Come abbiamo visto i services implementano servizi centralizzati per comunicare con le API.

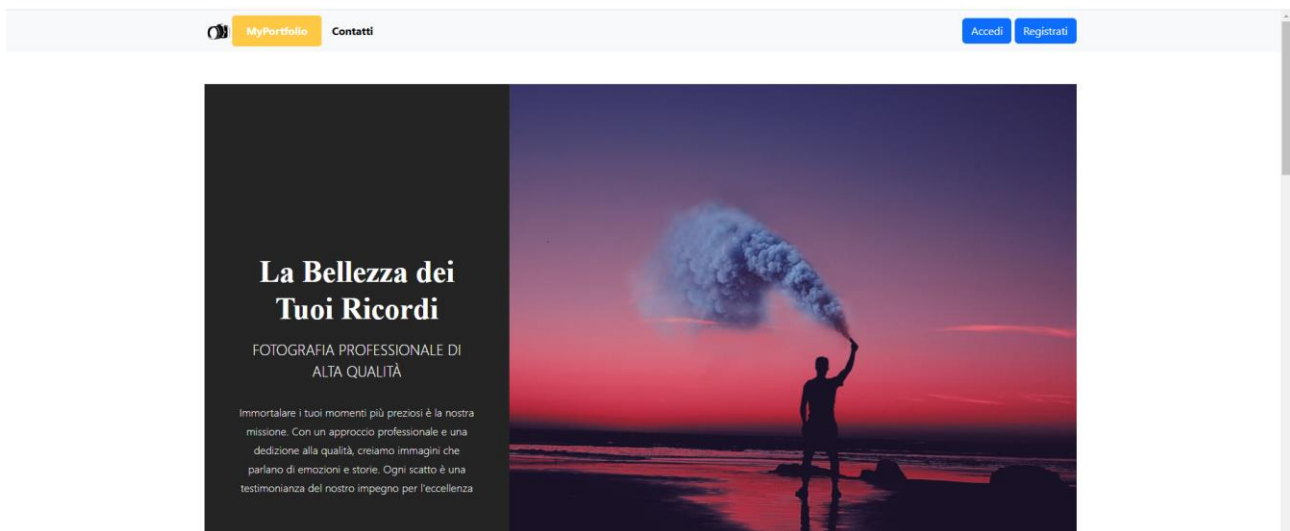
In questo esempio possiamo vedere la chiamata che permette di creare un nuovo ordine inviando una richiesta POST all'endpoint `api/shop` e la chiamata GET che permette di recuperare gli ordini fatti da un determinato utente chiamando l'endpoint `api/shop/myorder`

Index

La pagina di default su cui verranno indirizzati tutti gli utenti è l'*index*

In questa pagina sono presenti quattro sezioni principale:

- Il menu principale in cui sono presenti i pulsanti per andare sulla pagina contatti, il pulsante per effettuare il login e quello per la procedura di registrazione.
Nota: nella barra del menù dopo il login, appariranno pulsanti diversi a seconda del ruolo attribuito all'utente che effettua il login.
- una sezione dedicata alle Frequently Asked Questions
- una sezione Testimonials in cui l'admin potrà inserire alcune recensioni ricevute dai suoi client
- un footer (a comune a tutte le pagine) in cui sono presenti le informazioni di contatto, i Link ai social media, i link utili per la privacy policy, i termini di servizio e il supporto,



Contatti

La pagina *Contatti* è progettata per offrire agli utenti un punto di comunicazione diretto con l'amministratore del portfolio fotografico.

La pagina oltre a fornire le informazioni principali del proprietario del portfolio, svolge un ruolo essenziale nell'interazione tra gli utenti e il proprietario del portfolio, facilitando richieste di accesso ai lavori fotografici.

Funzionalità User

In questa sezione saranno riportate e descritte alcune delle funzionalità che può fare un'utenza con ruolo User, ovvero tutti gli utenti che si registrano sulla piattaforma.

Registrazione

Al momento della registrazione viene compilato lato backend la presenza di un'utenza già registrata con l'username e la mail indicata. Se il controllo dà esito positivo l'utenza viene registrata, altrimenti viene visualizzato un messaggio di errore.



Nome

test

Cognome

test

Username

test

Email

test@my-portfolio.it

Password

....

Registrati

Login

Classica form di login, che permette di effettuare il login e ricevere il token JWT da utilizzare nelle chiamate successive al back-end.

Ovviamente lato back-end, prima di inviare il token in risposta, viene verificata la presenza di un'utenza con l'username inserito e che la password inserita sia corretta.



Username

test

Password

....

Accedi

Gestione profilo

Da questa pagina gli utenti autenticati possono procedere in autonomia alla modifica dei dati personale come nome, cognome e indirizzo mail e alla modifica della password di accesso.

Lato back-end viene verificato che la password inserita sia corretta e che la nuova password sia diversa dalla vecchia.

Profilo

Info

Nome

Cognome

Email

Salva

Password

Vecchia password

Nuova password

Salva

Visualizzazione Gallerie fotografiche

La sezione più importante di questo progetto è la possibilità di visualizzare i lavori fotografici.

Gli utenti autenticati possono visualizzare i Work per i quali hanno diritto di visualizzazione recandosi alla pagina Gallery, in questa pagina dal dropdown menu possono scegliere il Work da visualizzare



Galleria

[Home](#) > Galleria

Photo Gallery

Benvenuti nella mia galleria fotografica, un viaggio visivo attraverso emozioni, luoghi e momenti catturati con passione e professionalità.
Se vuoi vedere altri lavori non esitare a contattarmi

Selezionando un Work appariranno le miniature delle immagini contenute all'interno. Questo garantisce un caricamento più veloce e fluido della pagina

Bird



Cliccando su un'immagine verrà richiesta al back-end l'immagine a dimensioni standard e mostrata all'interno di un pop-up.

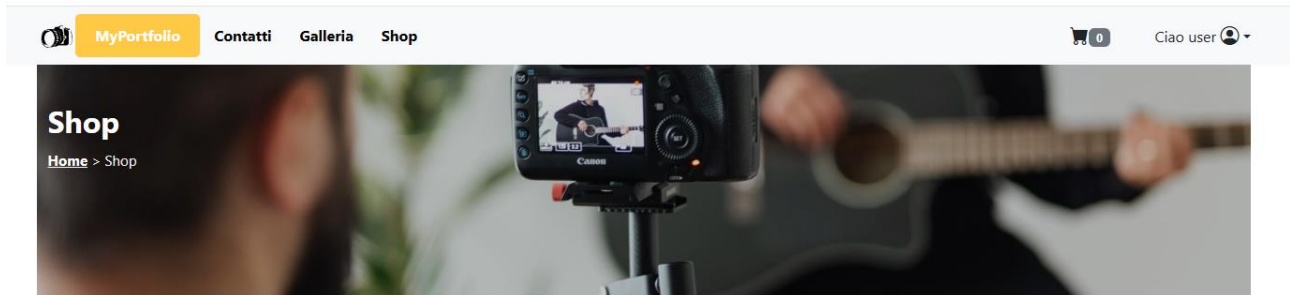


Come è possibile notare negli screenshot precedenti sia le miniature che le immagini a dimensione standard sono protette da copyright per evitare riusi non autorizzati.

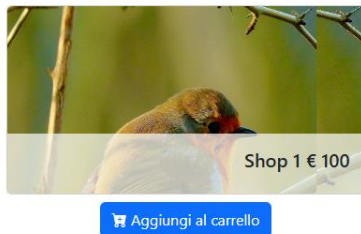
Shop

Nella sezione shop gli utenti autenticati posso acquistare alcune immagini che l'amministratore della piattaforma ha deciso di mettere in vendita.

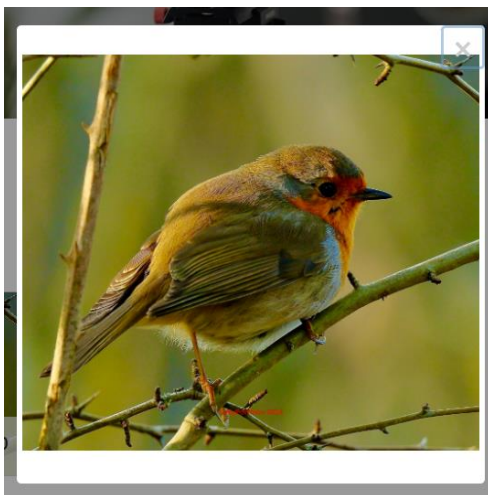
Le immagini sono considerate opere e possono essere acquistate in numero unico.



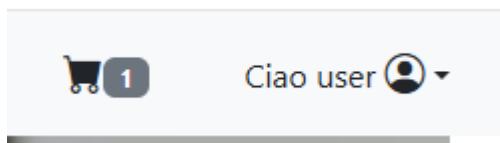
Shop



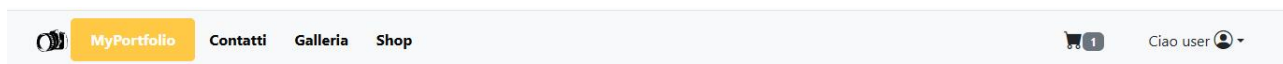
La visualizzazione ricalca le modalità di visualizzazione della Gallery, ovvero visualizzazione delle miniature al caricamento della pagina e visualizzazione di immagine a dimensioni standard a richiesta con pop-up



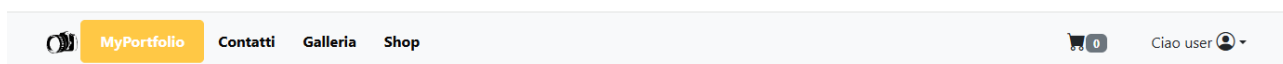
All'aggiunta del carrello viene anche incrementato il contatore dei prodotti nel carrello



Cliccando sull'icona del carrello si entra nella pagina Cart, dove è possibile rimuovere i prodotti dal carrello o procedere con l'acquisto.

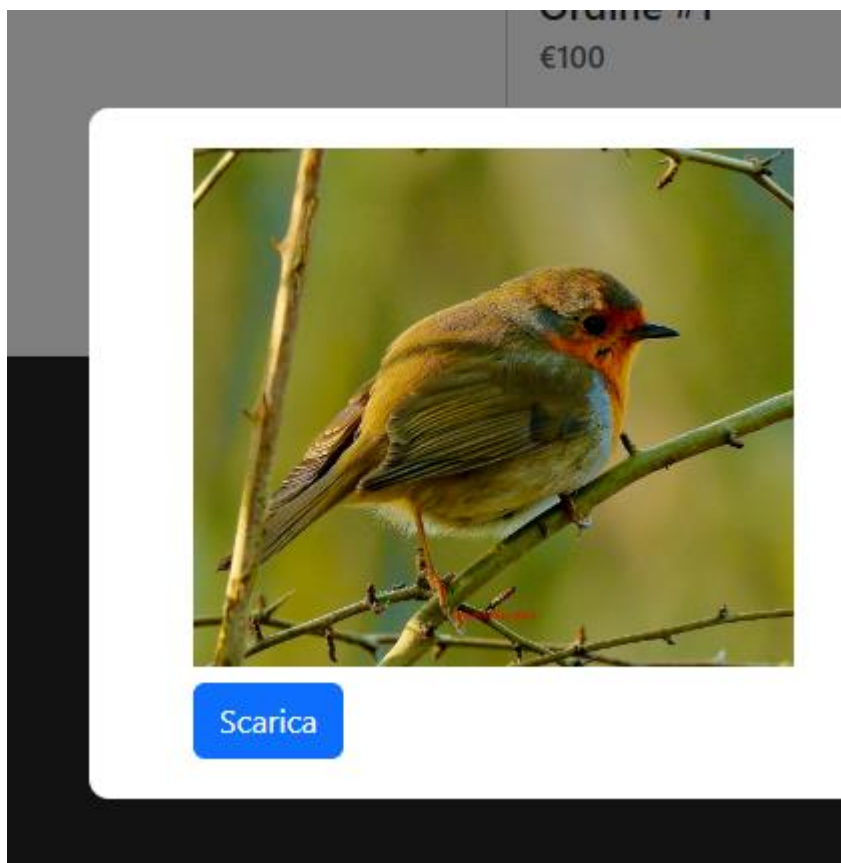


Dal menu a destra è possibile entrare nella pagina Orders, in cui è possibile visualizzare i propri ordini. Entrando nei dettagli di un ordine è possibile visualizzare tutte le immagini acquistate e procedere con il download dell'immagine



I miei ordini

Ordine #1 €100 23/11/2024 20:24 Dettaglio	Ordine #2 €100 04/12/2024 14:40 Dettaglio	Ordine #3 €1000 04/12/2024 14:43 Dettaglio
---	---	--

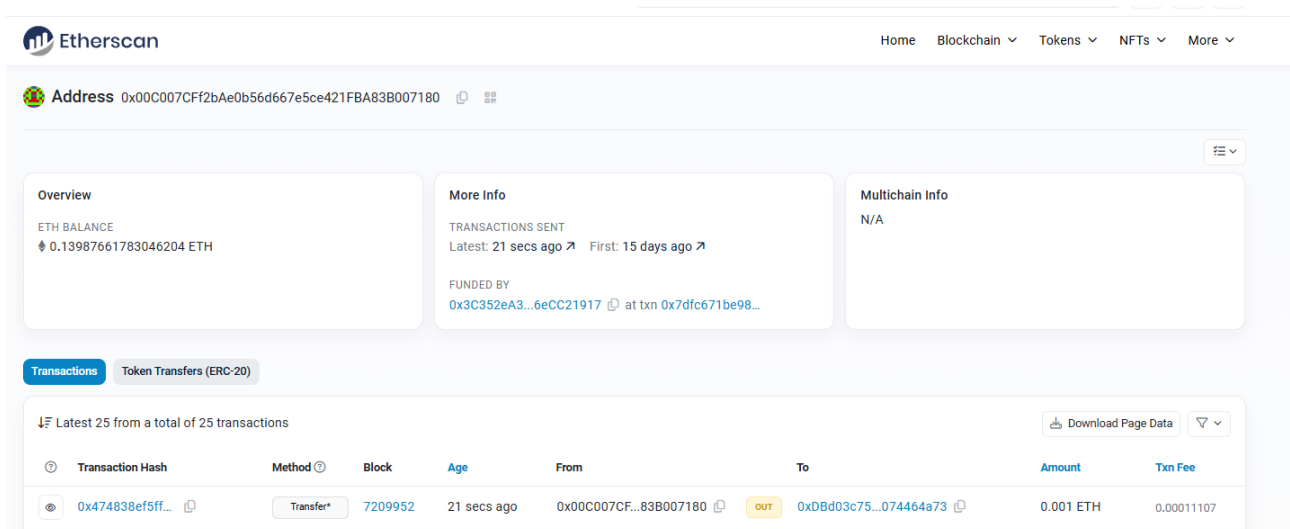


Nello screenshot sotto possiamo vedere lo stesso ordine, ma visto dalla visualizzazione Admin. In più possiamo vedere che ogni ordine fatto viene registrato come nuovo block sulla blockchain Ethereum e al momento della registrazione viene salvato il valore hash del nuovo blocco.

Ordine #3
€1000
04/12/2024 14:43

0x474838ef5ff8660d4e22cdf739593e
61e737abccedd5e3e78f0fc9aea38fe0
1e

[Dettaglio](#)



The screenshot shows the Etherscan interface for an address. The address is 0x00C007CFf2bAe0b56d667e5ce421FBA83B007180. The overview section shows an ETH balance of 0.13987661783046204 ETH. The more info section shows transactions sent, with the latest at 21 seconds ago and the first 15 days ago. The multichain info section is empty. The transactions section shows a list of transactions, with the latest one being a transfer of 0.001 ETH to 0xDBd03c75...074464a73.

Address: 0x00C007CFf2bAe0b56d667e5ce421FBA83B007180

Overview

ETH BALANCE
0.13987661783046204 ETH

More Info

TRANSACTIONS SENT
Latest: 21 secs ago First: 15 days ago

FUNDED BY
0x3C352eA3...6eCC21917 at txn 0x7dfc671be98...

Multichain Info
N/A

Transactions Token Transfers (ERC-20)

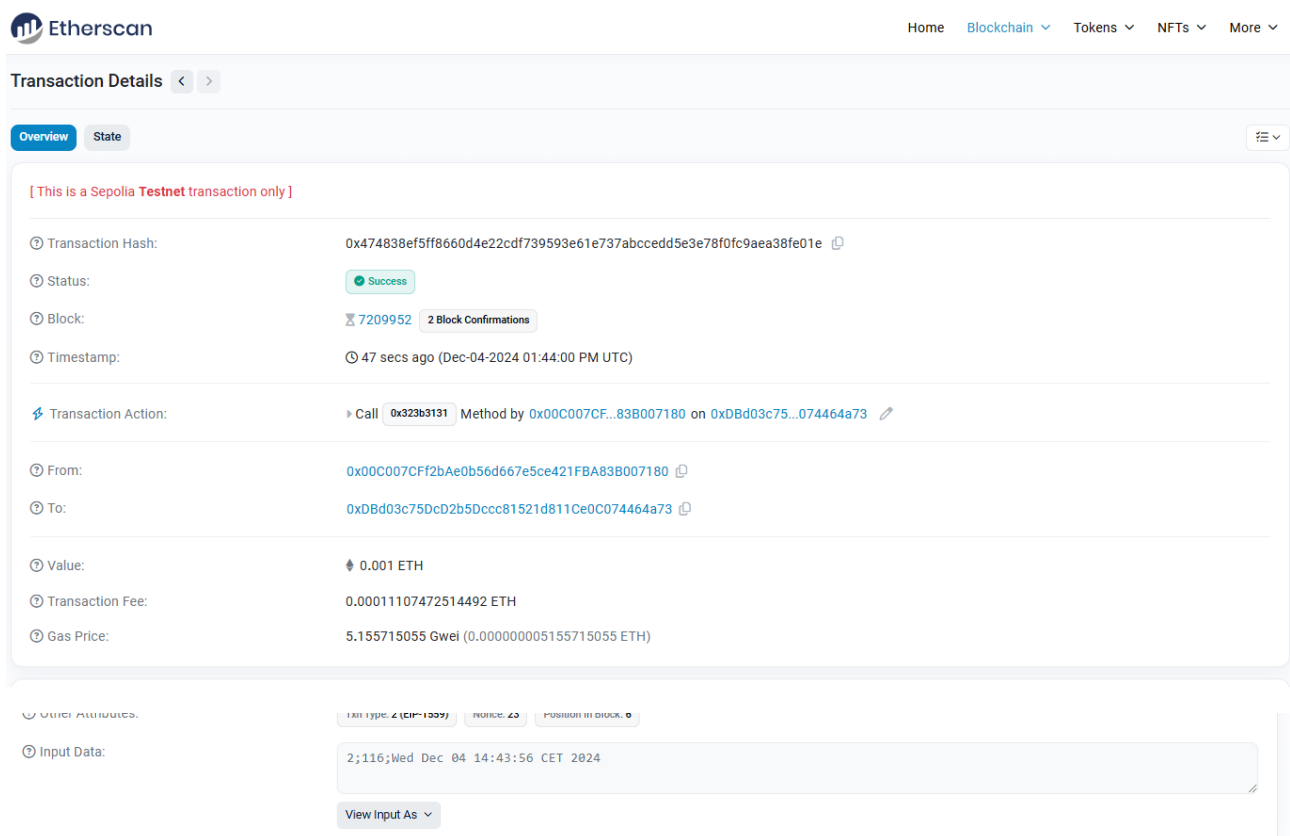
Latest 25 from a total of 25 transactions

Transaction Hash	Method	Block	Age	From	To	Amount	Txn Fee
0x474838ef5ff...	Transfer*	7209952	21 secs ago	0x00C007CF...83B007180	0xDBd03c75...074464a73	0.001 ETH	0.00011107

<https://sepolia.etherscan.io/address/0x00C007CFf2bAe0b56d667e5ce421FBA83B007180>

Possiamo notare che la transazione viene registrata all'interno della net Ethereum.

Se espandiamo i dettagli possiamo vedere tutti i parametri della transazione. Nel campo input data possiamo notare l'id dell'utente che ha fatto l'acquisto, l'immagine o le immagini acquistate e il timestamp dell'acquisto.



The screenshot shows the transaction details page for the transaction 0x474838ef5ff8660d4e22cdf739593e61e737abccedd5e3e78f0fc9aea38fe01e. The transaction is a call to 0x32b3131 by 0x00C007CF...83B007180 on 0xDBd03c75...074464a73. The transaction is successful and has 2 block confirmations. The value is 0.001 ETH and the transaction fee is 0.00011107472514492 ETH. The gas price is 5.155715055 Gwei.

Transaction Details

Overview State

[This is a Sepolia Testnet transaction only]

Transaction Hash: 0x474838ef5ff8660d4e22cdf739593e61e737abccedd5e3e78f0fc9aea38fe01e

Status: Success

Block: 7209952 2 Block Confirmations

Timestamp: 47 secs ago (Dec-04-2024 01:44:00 PM UTC)

Transaction Action: Call 0x32b3131 Method by 0x00C007CF...83B007180 on 0xDBd03c75...074464a73

From: 0x00C007CFf2bAe0b56d667e5ce421FBA83B007180

To: 0xDBd03c75DcD2b5Dccc81521d811Ce0C074464a73

Value: 0.001 ETH

Transaction Fee: 0.00011107472514492 ETH

Gas Price: 5.155715055 Gwei (0.000000005155715055 ETH)

Other Attributes:

Txn type: 2 (CALL) nonce: 43 position in block: 0

Input Data: 2;116;Wed Dec 04 14:43:56 CET 2024

View Input As

<https://sepolia.etherscan.io/tx/0x474838ef5ff8660d4e22cdf739593e61e737abccedd5e3e78f0fc9aea38fe01e>

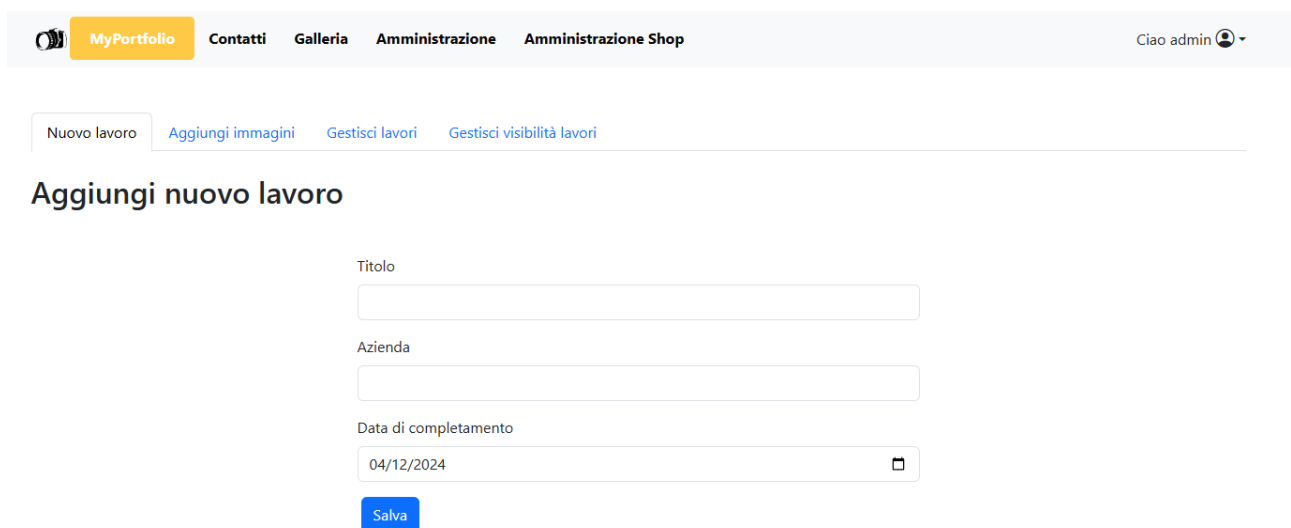
Funzionalità Admin

In questa sezione saranno riportate e descritte alcune delle funzionalità che può fare un'utenza con ruolo Admin. Vedremo prima la creazione e gestione dei Work le relative immagini contenute, passeremo poi a vedere le funzionalità relative alla gestione dello Shop.

Inserimento nuovo lavoro

Il primo passo per poter mostrare i propri lavori agli utenti è quello di creare un Lavoro (Work), che conterrà al suo interno le immagini.

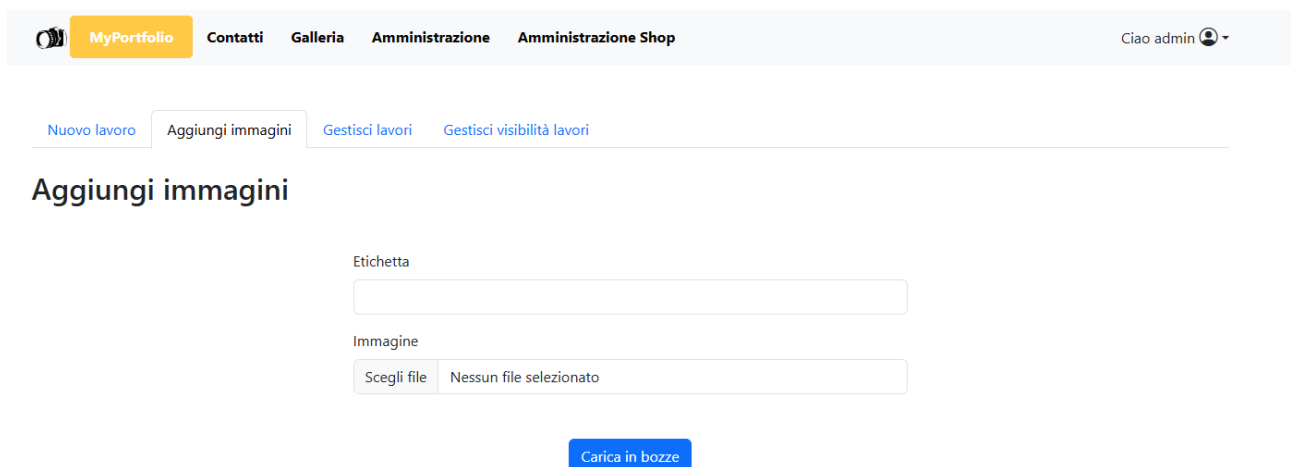
Per la creazione di un lavoro è richiesto di indicare il titolo, l'azienda o cliente per il quale si è svolto quel lavoro e la data di completamento



The screenshot shows the 'Nuovo lavoro' (New work) form in the Admin interface. The top navigation bar includes 'MyPortfolio', 'Contatti', 'Galleria', 'Amministrazione', and 'Amministrazione Shop'. The user is logged in as 'Ciao admin'. Below the navigation bar, there are four tabs: 'Nuovo lavoro' (selected), 'Aggiungi immagini', 'Gestisci lavori', and 'Gestisci visibilità lavori'. The form itself has three input fields: 'Titolo' (Title), 'Azienda' (Company), and 'Data di completamento' (Completion date). The 'Data di completamento' field is pre-filled with '04/12/2024'. A blue 'Salva' (Save) button is at the bottom of the form.

Aggiunta immagini ad un Work



Nella sezione denominata "Aggiungi immagini" è possibile caricare e salvare all'interno del database l'immagini



The screenshot shows the 'Aggiungi immagini' (Add images) form in the Admin interface. The top navigation bar is the same as the previous screenshot. Below the navigation bar, there are four tabs: 'Nuovo lavoro', 'Aggiungi immagini' (selected), 'Gestisci lavori', and 'Gestisci visibilità lavori'. The form has two input fields: 'Etichetta' (Label) and 'Immagine' (Image). The 'Immagine' field is pre-filled with 'Scegli file' (Choose file) and 'Nessun file selezionato' (No file selected). A blue 'Carica in bozze' (Upload as draft) button is at the bottom of the form.

Le immagini vengono caricate una per volta e al click del tasto, vengono messe nella tabella di bozze per eventuali modifiche o rimozione prima di essere caricate sul DB.

Si ricorda che le immagini non vengono memorizzate come blob sul database, ma viene salvato solamente l'url del NAS.

 **MyPortfolio** [Contatti](#) [Galleria](#) [Amministrazione](#) [Amministrazione Shop](#) Ciao admin 

[Nuovo lavoro](#) [Aggiungi immagini](#) [Gestisci lavori](#) [Gestisci visibilità lavori](#)

Aggiungi immagini



Etichetta

Immagine

Scegli file

438159401-Prothonotary_Warbler-Ryan_Justice-social.jpg

Carica in bozze

Etichetta	URL	URL Thumbnail	
Bird 3	<input type="text" value="C:/Users/Andrea/Pictures/"/>	C:/Users/Andrea/Pictures/my-portfolio/bird/thumb/438159401-Prothonotary_Warbler-Ryan_Justice-social.jpg	
Bird			 <div>Salva</div>

Poiché per ragioni di sicurezza non è consentito recuperare l'url locale di un file, in quanto consentire a un sito web di accedere al percorso completo dei file locali dell'utente potrebbe esporre informazioni sensibili sulla struttura del file system dell'utente, il campo url è una text box modificabile. L'url viene composto dalla variabile BASE URL dichiarata globalmente nei file di configurazione aggiungendo il nome del file. Se il file non dovesse essere nella BASE URL allora l'admin procederà a modificare l'url in modo che corrisponda alla posizione esatta in cui si trova l'immagine. In automatico viene aggiornata l'URL della miniatura.

Una volta completata la tabella di bozza è possibile scegliere a quale Work associare le immagini e cliccando su salva verrà memorizzato tutto sul Database.

Gestione Work e immagini contenute

Dalla sezione "Gestisci lavori" è possibile scegliere tramite il menu quale Work modificare

 **MyPortfolio** [Contatti](#) [Galleria](#) [Amministrazione](#) [Amministrazione Shop](#) Ciao admin 

[Nuovo lavoro](#) [Aggiungi immagini](#) [Gestisci lavori](#) [Gestisci visibilità lavori](#)

Gestisci lavori

Sarà possibile modificare sia i campi Titolo, Azienda e Data di complemento.

Nuovo lavoro Aggiungi immagini Gestisci lavori Gestisci visibilità lavori

Gestisci lavori

Bird

Aggiorna lavoro

Titolo

Bird

Azienda

Acme corp.

Data di completamento

04/12/2024

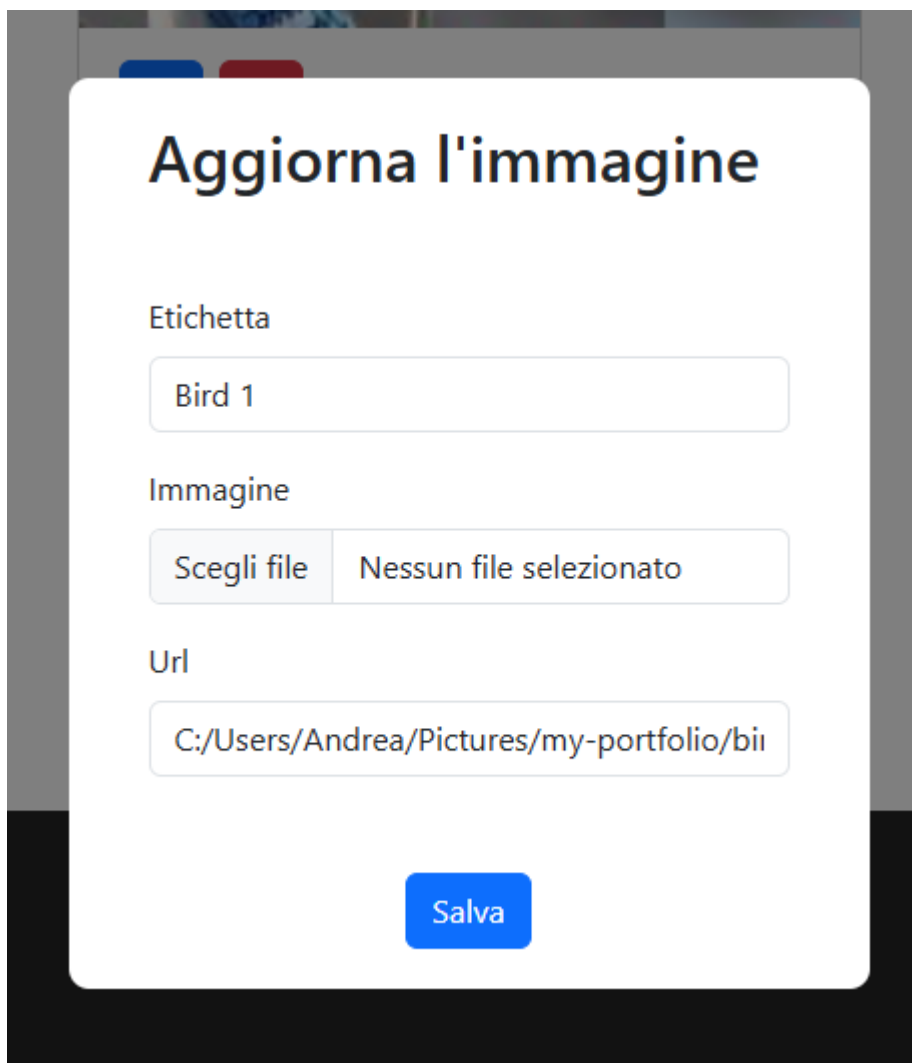
Salva

Elimina



Ma anche cancellare o modificare i campi delle singole immagini al suo interno.





Aggiorna l'immagine

Etichetta

Immagine


Scegli file Nessun file selezionato

Url

Salva

Gestione visibilità work

Nella sezione “Gestisci visibilità lavori” selezionando un utente dal menu a tendina



MyPortfolio Contatti Galleria Amministrazione Amministrazione Shop Ciao admin

Nuovo lavoro Aggiungi immagini Gestisci lavori Gestisci visibilità lavori

Gestisci la visibilità dei lavori

Salva

Apparirà una tabella nella quale è possibile concedere o revocare i diritti di visualizzazione dei lavori.

Se l'utente ha già i diritti di visualizzazione su un determinato lavoro la colonna Visibile avrà già la spunta.

[Nuovo lavoro](#) [Aggiungi immagini](#) [Gestisci lavori](#) [Gestisci visibilità lavori](#)

Gestisci la visibilità dei lavori

user user

Salva

Titolo	Azienda	Data di completamento	Visibile
Bird	Acme corp.	04/12/2024	<input type="checkbox"/>

[Nuovo lavoro](#) [Aggiungi immagini](#) [Gestisci lavori](#) [Gestisci visibilità lavori](#)

Gestisci la visibilità dei lavori

user user

Salva

Titolo	Azienda	Data di completamento	Visibile
Bird	Acme corp.	04/12/2024	<input checked="" type="checkbox"/>

Gestione immagine nella sezione shop

Nella pagina Amministrazione Shop è possibile gestire caricare nuove immagini da mettere in vendita nello shop.

Gestione Shop

[Gestisci immagini](#) [Ordini](#) [Verifica hash](#)

Immagini



Nessuna immagine trovata

Carica la tua prima immagine



Gestione Shop

[Gestisci immagini](#)[Ordini](#)[Verifica hash](#)

Immagini



Shop 1 € 100



È possibile modificare o eliminare le immagini dallo shop.

Carica una nuova immagine

Etichetta

Prezzo

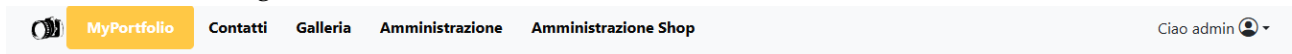
Immagine

Scegli file

Carica

Storico ordine effettuati

Nella sezione ordini è possibile visualizzare tutti gli ordini fatti all'interno dello store con il valore hash associato ad ognuno



Gestione Shop

[Gestisci immagini](#) [Ordini](#) [Verifica hash](#)

Ordini

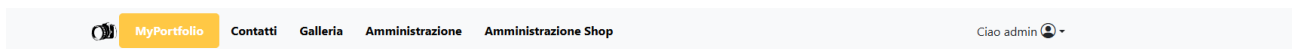
Ordine #1
€100
23/11/2024 20:24

0xd34efebd7fe72ea3b240fd88b5fc9a
3677dba6f268a8598f49241b24d0f0a
3c2

[Dettaglio](#)

Verifica HASH ordini

È possibile inviare una richiesta di verifica Hash sulla rete Ethereum. La risposta di validità apparirà in un pop up in basso a destra.



Gestione Shop

[Gestisci immagini](#) [Ordini](#) [Verifica hash](#)

Verifica hash

Hash

Verifica

Contattaci



Marco Rossi
Via Roma, 10,
00184 Roma RM

Lun - Ven: 9 - 17

Telefono
+39 06 1234567

E-mail
marco.rossi@my-portfolio.com

Support Links

[Privacy Policy](#)
[Terms of Service](#)
[Contact us](#)
[Support](#)



Sezione shop e blockchain

Nota: questa funzionalità è stata sviluppata come estensione al progetto di base (che rimarrà solo per scopi didattici, almeno per il momento), inserita sia per sviluppare nuove funzionalità, sia come mezzo di studio

per analizzare, studiare e implementare la possibilità di integrare e certificare gli acquisti all'interno di una blockchain.

Come visto nella parte introduttiva, oltre alla gestione delle gallerie fotografiche il sistema include una sezione shop, dove l'amministratore può caricare immagini in vendita, denominate ShopableImage, disponibili in edizione unica (opere fotografiche a tiratura unica). Gli utenti registrati possono acquistare queste immagini. Ogni transazione (acquisto) viene registrata come un blocco su una blockchain Ethereum tramite l'integrazione della piattaforma Infura. Questa soluzione garantisce la sicurezza e la tracciabilità di ogni transazione, conferendo unicità agli acquisti effettuati all'interno dell'applicazione.

Prima di trovare e testare la piattaforma Infura, si erano analizzate alcune repo di GitHub che vedono l'esecuzione di un nodo locale e/o l'utilizzo di librerie che gestiscono una blockchain locale.

La decisione è poi passata all'utilizzo di Infura (anche se a pagamento fornisce un buon numero di call API giornaliere gratuite, che per una fase di test sono sufficienti) che presenta diversi vantaggi, soprattutto in termini di praticità, efficienza e scalabilità.

I principali vantaggi dell'uso di una net remota sono:

- con l'uso di un nodo locale è necessario scaricare l'intera blockchain (nel caso di Ethereum, può richiedere centinaia di GB di spazio), configurarlo e mantenerlo aggiornato. Questo implica anche monitoraggio costante e gestione delle risorse del server, oltre a potenziali problemi di downtime o aggiornamenti di sicurezza.
- È necessario garantire che la rete sia sempre attiva, sincronizzata e in grado di rispondere rapidamente alle richieste.
- Un nodo locale potrebbe funzionare bene per progetti più piccoli o durante lo sviluppo, ma scalare verso un'applicazione di produzione con molte richieste potrebbe richiedere l'esecuzione di più nodi, bilanciamento del carico e altre complessità legate all'infrastruttura.
- Eseguire un nodo Ethereum (o anche una Net ex-novo) richiede una macchina server dedicata con spazio di archiviazione considerevole, RAM, CPU e connessione Internet affidabile.
- Un nodo locale richiede monitoraggio costante per assicurarsi che sia aggiornato e sicuro. Per evitare il rischio di exploit o vulnerabilità.

Descrizione piattaforma

Infura è una piattaforma che fornisce accesso a diverse blockchain, permettendo agli sviluppatori di interagire con queste reti senza dover configurare un nodo completo. È particolarmente utilizzata per Ethereum, ma supporta anche altre blockchain come Polygon, Optimism, Arbitrum e IPFS.

La piattaforma fornisce un importante aiuto per mettere in comunicazione le applicazioni con le reti blockchain, fornendo agli sviluppatori endpoint API che gli permettano di interagire in modo facile con le principali blockchain.

Nel dettaglio la piattaforma di base un'infrastruttura cloud che permette agli sviluppatori di connettersi facilmente a blockchain distribuite tramite delle API HTTP e WebSocket. Questo elimina la necessità di gestire e mantenere un nodo blockchain localmente, risparmiando costi e complessità di configurazione.

Network e principale e fork utilizzati

Infura si appoggia principalmente alla **blockchain di Ethereum**, fornendo nodi per le reti principali e per le testnet come Rinkeby, Ropsten, Kovan e Goerli. Tuttavia, come menzionato, supporta anche altre reti come Polygon, Optimism, Arbitrum e IPFS per fornire maggiore interoperabilità tra diverse blockchain e tecnologie decentralizzate.

Per questo progetto (anche per la sua fama) si è scelto di utilizzare la rete Ethereum che è una rete blockchain decentralizzata, open source e Turing-complete.

Questa net può essere utilizzata per effettuare/registrare transazioni e comunicare senza essere controllato da un'autorità centrale, utilizzando il meccanismo di consenso Proof of Stake (PoS).

Su questa Net possono essere utilizzate queste 2 fork di test:

Holesky JSON-RPC over HTTPS <https://holesky.infura.io/v3/<API-KEY>>

Sepolia JSON-RPC over HTTPS <https://sepolia.infura.io/v3/<API-KEY>>

Le testnet **Holesky** e **Sepolia** sono entrambe utilizzabili per effettuare test su Ethereum in un ambiente sicuro, ma presentano alcune differenze fondamentali in termini di scopo, utilizzo, dimensioni e architettura.

- **Holesky**: è progettata per gestire grandi volumi di validatori e replicare fedelmente le condizioni della mainnet, rendendola ideale per test su larga scala. Ha un numero molto elevato di validatori (oltre 1,4 milioni al suo lancio). Questo la rende una rete robusta e adatta a testare aggiornamenti della rete Ethereum, nuove funzionalità, e progetti che richiedono una simulazione più realistica delle condizioni della mainnet.
I token Holesky non hanno valore reale, gli utenti possono richiedere token di test per testare le loro applicazioni. Poiché i token sono abbondanti e gratuiti, è una testnet ideale per progetti di sviluppo intensivo.
- **Sepolia**: è una testnet utilizzata principalmente per test meno impegnativi e applicazioni decentralizzate (dApp). Non è pensata per simulare l'intera rete Ethereum in scala come Holesky, ma piuttosto per fornire un ambiente di sviluppo snello e stabile. Ha un numero di validatori molto inferiore rispetto a Holesky, e viene utilizzata per test più ridotti e mirati. Questo la rende ideale per sviluppatori che desiderano eseguire test più rapidi e con meno overhead.

Sono stati fatti test su entrambe le Net e si visto anche il numero ridotto di transizioni provate non si notano differenze significative.

Implementazione e flusso

Per capire i flussi per la registrazione di una transazione e la verifica di veridicità di un Hash si utilizzerà un approccio bottom-up: vedremo prima i due script python che si occupano delle chiamate API verso la piattaforma Infura, poi analizzeremo i controller e la classe di utilità utilizzate per ricevere ed elaborare le richieste che arrivano dal Front-End, sviluppando ed analizzando concetti e librerie specifiche della piattaforma.

Script python

```
from web3 import Web3, exceptions
import sys
import random
from eth_account import Account

account = Account.create()
nuovo_indirizzo = account.address

infura_url = 'https://sepolia.infura.io/v3/5cb88f299e974e9082c695c5fb3e9b13'
private_key = '#' #metamask
from_account = '0x00C007CFf2bAe0b56d667e5ce421FBA83B007180'
to_account=nuovo_indirizzo
web3 = Web3(Web3.HTTPProvider(infura_url))

try:
    from_account = web3.to_checksum_address(from_account)
except exceptions.InvalidAddress:
    print(f"Invalid 'from_account' address: {from_account}")

try:
    to_account = web3.to_checksum_address(to_account)
except exceptions.InvalidAddress:
    print(f"Invalid 'to_account' address: {to_account}")

data =sys.argv[1]+" "+"sys.argv[2]+" "+"sys.argv[3]
nonce = web3.eth.get_transaction_count(from_account,'pending');
gas_price = web3.eth.gas_price
base_fee = web3.eth.fee_history(1, "latest")['baseFeePerGas'][-1]
priority_fee = web3.to_wei(2, 'gwei')

tx = {
    #'type': '0x2',
    'nonce': nonce,
    'from': from_account,
    'to': to_account,
    'value': web3.to_wei(0.001, 'ether'),
    'chainId': 11155111,
    'data': data.encode("utf-8").hex(),
    'maxFeePerGas': base_fee + priority_fee,
    'maxPriorityFeePerGas': priority_fee,
    'gas': 32000
}

gas = web3.eth.estimate_gas(tx)
tx['gas'] = gas
signed_tx = web3.eth.account.sign_transaction(tx, private_key)
tx_hash = web3.eth.send_raw_transaction(signed_tx.rawTransaction)
print(str(web3.to_hex(tx_hash)))
```

Snippet 33: registry.py script scritto in Python per la registrazione di una nuova transizione

Il primo script che analizziamo è *registry.py* che si occupa di ricevere i dati da registrare dal back-end e di registrare una nuova transizione all'interno della Test Net (nel caso dello snippet Sepolia) utilizzando il framework **Web3.py**.

La transazione (essendo in ambiente di test) non trasferisce Ether, ma include un payload di dati (data), codificati come stringa esadecimale. Lo script genera inoltre un indirizzo casuale come destinatario, simula il gas, e firma la transazione con una chiave privata.

Vediamo i vari passi nel dettaglio

1. **Import:**

- **Web3:** Importa la libreria Web3 per interagire con la blockchain Ethereum.
- **exceptions:** Consente di gestire eccezioni specifiche di Web3, come indirizzi non validi.
- **sys:** Usato per accedere agli argomenti della riga di comando (sys.argv).
- **random:** Usato per generare un valore casuale per il destinatario.

2. **Configurazione dell'URL di Infura e delle chiavi Metamask:**

- **infura_url:** Definisce l'endpoint per connettersi alla testnet **Holesky** tramite Infura.
- **private_key:** È la chiave privata dell'account Metamask che firmerà la transazione.
- **from_account:** Contiene l'indirizzo dell'account che invia la transazione

3. **Generazione casuale del destinatario:**

- L'indirizzo di destinazione viene generato in modo randomico sfruttando la funzione Account.create()

4. **Connessione alla blockchain tramite Infura:**

- **web3:** Crea una connessione alla blockchain Ethereum usando l'URL Infura specificato.

5. **Validazione degli indirizzi:**

- **from_account** e **to_account** vengono convertiti nel formato **checksum** (uno standard per gli indirizzi Ethereum che rileva gli errori di battitura).
- Se uno degli indirizzi è invalido, l'eccezione viene catturata e un messaggio di errore viene mostrato.

6. **Costruzione del payload di dati:**

- Combina i primi tre argomenti della riga di comando in una stringa, separati da un punto e virgola (;). Questa stringa verrà inclusa come dati nella transazione.

7. **Recupero del nonce:**

- Il **nonce** rappresenta il numero di transazioni inviate dall'account mittente. Ogni transazione deve avere un nonce univoco per evitare conflitti. NOTA: nel metodo è necessario passare il parametro *pending* per avere il nonce di tutti i blocchi, anche di quelli non ancora verificati.

8. Creazione della transazione:

- **type:** Tipo di transazione (0x2 per le transazioni EIP-1559 su Ethereum).
- **nonce:** Numero di transazioni precedenti inviate dall'account.
- **from:** Indirizzo mittente.
- **to:** Indirizzo destinatario
- **value:** La quantità di Ether trasferita è 0, poiché l'obiettivo è inviare solo i dati.
- **maxFeePerGas** e **maxPriorityFeePerGas:** Impostati a 0 gwei, quindi nessun costo di gas verrà pagato ai validatori del blocco.
- **chainId:** L'ID della rete (17000 corrisponde alla rete Holesky).
- **data:** I dati da includere nella transazione vengono codificati in esadecimale con utf-8

9. Firma della transazione:

- La transazione viene firmata utilizzando la chiave privata dell'account mittente.

10. Invio della transazione:

- La transazione firmata viene inviata alla rete Ethereum.
- L'hash della transazione viene stampato in formato esadecimale, consentendo di monitorare la transazione sulla blockchain.

```
import sys
from web3 import Web3

infura_url = 'https://holesky.infura.io/v3/5cb88f299e974e9082c695c5fb3e9b13'
web3 = Web3(Web3.HTTPProvider(infura_url))
print(web3.eth.get_transaction(sys.argv[1]))
```

Snippet 34:hash_verify.py script scritto in Python per la verifica di una transazione

Vediamo il funzionamento nel dettaglio:

1. Import:

- **Web3:** Importa la libreria Web3 per interagire con la blockchain Ethereum.
- **sys:** Usato per accedere agli argomenti della riga di comando (sys.argv).

2. Esecuzione:

- Viene creata un'istanza di Web3, usando HTTPProvider per connettersi alla rete tramite l'endpoint Infura
- Si recupera l'hash della transazione da analizzare (passata come parametro).
- Si richiede alla rete Ethereum le informazioni relative alla transazione il cui hash è fornito come input. La funzione restituisce i dettagli completi della transazione, come mittente, destinatario, valore, gas, ecc.

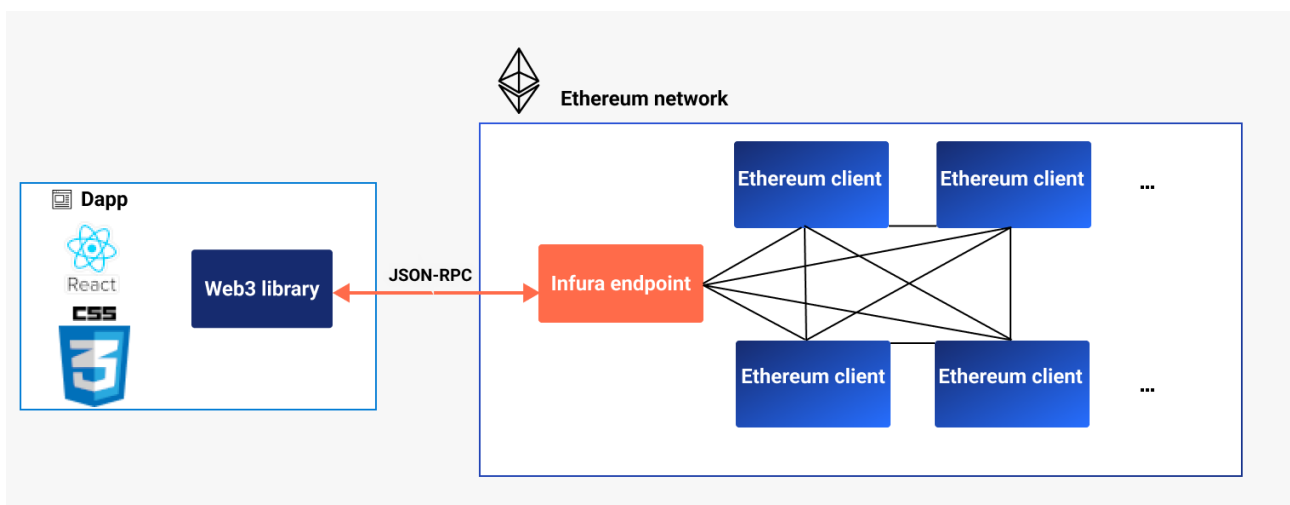
Libreria Web3 e transaction type

Vediamo adesso alcuni dettagli relativi alla libreria Web3 e la transizione `eth.estimate_gas` utilizzata.

Le librerie Web3 forniscono l'accesso a metodi di supporto che consentono di interagire con la blockchain tramite un provider Web3 come Infura.

Le librerie Web3 consentono la creazione di moduli in grado di comunicare con la blockchain, in particolare possono essere utilizzate per inviare transazioni, leggere dati dagli smart contract e interagire con la blockchain

L'immagine seguente mostra come una libreria Web3 può connettersi al nodo di una Net (in questo esempio Infura) per comunicare con la blockchain.



La nostra app o meglio decentralized app (dapp), con l'uso delle librerie Web3 e del protocollo **JSON-RPC** (metodo che permette a un client di invocare funzioni in esecuzione su un server remoto. Nel contesto di Ethereum, questo protocollo viene utilizzato per inviare richieste come "leggere il saldo di un indirizzo" o "trasmettere una transazione"), ci permette di inserire una transazione o verificare una transazione in maniera molto semplice e del tutto trasparente rispetto all'architettura della Net.

Nel caso di questo progetto si è deciso di provare ad utilizzare la libreria **web3.py** per interagire con Ethereum [<https://web3py.readthedocs.io/en/stable/>].

La tipologia di transazione utilizzata è `eth.estimate_gas` che genera e restituisce una stima di quanto gas è necessario per consentire il completamento della transazione. La transazione non verrà aggiunta alla blockchain, ma restituisce comunque un hash.

Il gas è la quantità di **calcolo** necessaria per eseguire operazioni sulla rete Ethereum. Ogni operazione richiede una certa quantità di calcolo da parte dei nodi della rete. Il gas viene utilizzato per compensare i miner (o i validatori, in caso di Ethereum 2.0) che forniscono la potenza computazionale per eseguire queste operazioni

Il gas esiste per garantire che:

- I miner/validatori siano pagati per il calcolo che forniscono.

- Si eviti l'esecuzione di transazioni o contratti inefficienti o malevoli. Poiché il gas deve essere pagato, gli utenti non possono abusare del sistema eseguendo operazioni che richiedono troppo calcolo o creano loop infiniti.

Componenti lato Back-End

Lato back-end *it.myportfolio.utility.BlockchainTransactionService* sono presenti due metodi: il primo che analizzeremo permette la registrazione di una nuova transizione:

```
public static String registry_transaction(Long UserID, List<Long> ImageID, Date
timestamp) {

    try {
        // Percorso dello script Python da eseguire
        String scriptPath = "registry.py";

        // Parametri da passare allo script Python
        String UID = UserID.toString();
        String IID="";
        for (Long id : ImageID) {
            IID=IID+id.toString()+" ";
        }

        String date = timestamp.toString();

        ProcessBuilder pb = new ProcessBuilder("python",
            scriptPath, UID, IID, date);

        // Avvia il processo
        Process process = pb.start();

        // Ottieni l'output del processo
        BufferedReader reader = new BufferedReader(new
            InputStreamReader(process.getInputStream()));
        String line = reader.readLine();
        System.out.println(line);

        // read any errors from the attempted command
        BufferedReader stdError = new BufferedReader(
            new InputStreamReader(process.getErrorStream()));
        String s;
        while ((s = stdError.readLine()) != null) {
            System.out.println(s);
        }

        // Attendere il termine dell'esecuzione dello script
        int exitCode = process.waitFor();
        if (exitCode == 0) {
            return line;
        }

    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
    }
    return null;
}
```

Snippet 35: metodo per la registrazione di una nuova transizione all'interno della TestNet

- **Parametri di input:** il metodo riceve tre parametri:
 - **UserID** (ID dell'utente)
 - **ImageID** (lista di ID delle immagini)
 - **timestamp** (data e ora della transazione).
- **Preparazione dei parametri:** Converte l'ID utente e gli ID delle immagini in stringhe. Gli ID delle immagini vengono concatenati in un'unica stringa separata da virgole.
- **Esecuzione dello script Python:** viene creato un oggetto ProcessBuilder che costruisce un comando per eseguire lo script Python (registry.py) con i parametri.
- **Lettura dell'output:** il metodo legge l'output dello script Python (attraverso uno stream), viene catturato e l'eventuale errore generato dall'esecuzione del processo.
- **Ritorno dell'output:** Se lo script Python viene eseguito con successo (exit code 0), l'output principale viene restituito come stringa. In caso di errore, viene restituito null.

```
public static String verify_hash(String Hash) {

    try {
        // Percorso dello script Python da eseguire
        String scriptPath = "hash_verify.py";

        // Costruisci il comando da eseguire
        ProcessBuilder pb = new ProcessBuilder("python", scriptPath, Hash);

        // Avvia il processo
        Process process = pb.start();

        // Ottieni l'output del processo
        BufferedReader reader = new BufferedReader(new
            InputStreamReader(process.getInputStream()));
        String line = reader.readLine();
        System.out.println(line);

        BufferedReader stdError = new BufferedReader(new
            InputStreamReader(process.getErrorStream()));
        String s;
        System.out.println("Here is the standard error of
                           the command (if any):\n");
        while ((s = stdError.readLine()) != null) {
            System.out.println(s);
        }

        // Attendere il termine dell'esecuzione dello script
        int exitCode = process.waitFor();
        if (exitCode == 0) {
            return line;
        }

    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
    }
    return null;
}
```

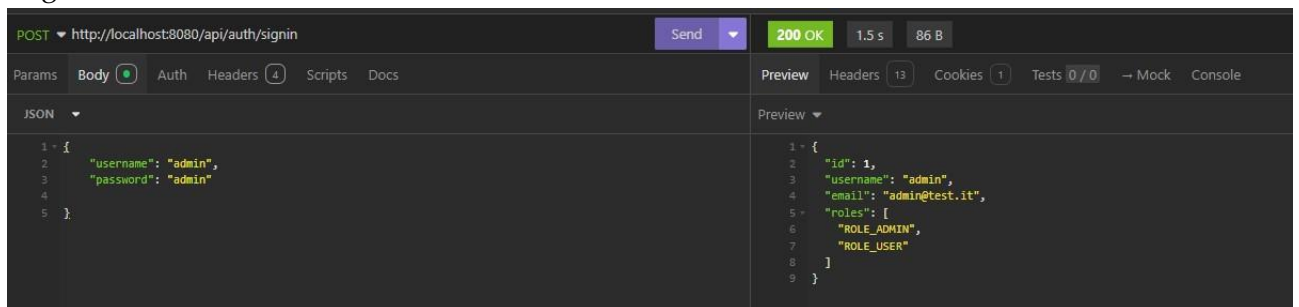
Snippet 36: metodo per la verifica di una transizione all'interno della TestNet

In modo del tutto equivalente, tramite: viene creato un oggetto `ProcessBuilder` che costruisce un comando per eseguire lo script Python (`hash_verify.py`) con l'hash in input. Il metodo legge l'output dello script Python (attraverso uno stream), viene catturato e l'eventuale errore generato dall'esecuzione del processo.

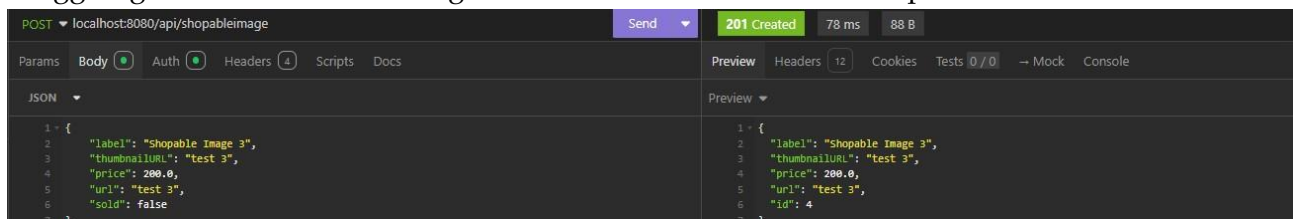
Se lo script Python viene eseguito con successo (exit code 0), l'output principale viene restituito come stringa. In caso di errore, viene restituito null.

Per concludere questa sezione si riporta un test completo simulando con Insomnia l'acquisto di opere, la registrazione dell'acquisto fino alla verifica dell'hash del blocco.

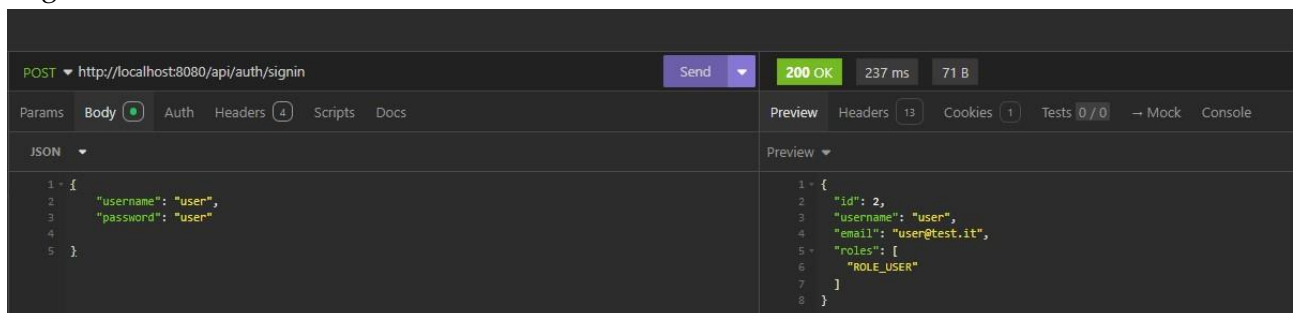
1. Login con utenza admin, in modo da ottenere il token bearer



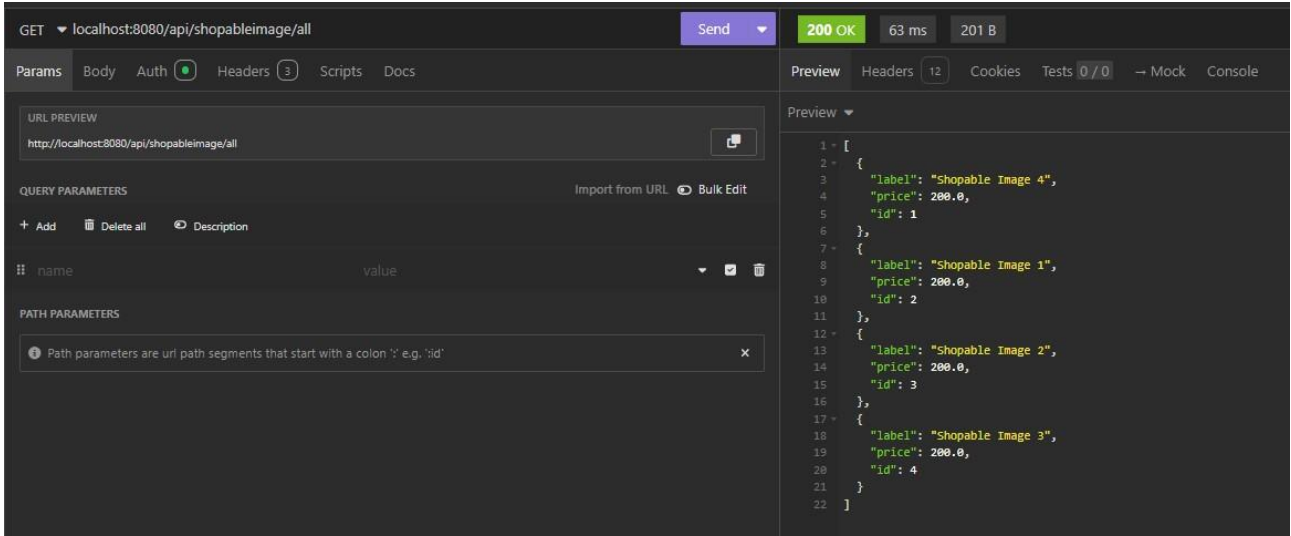
2. Si aggiungono al DB alcune immagini da mettere in vendita nello shop



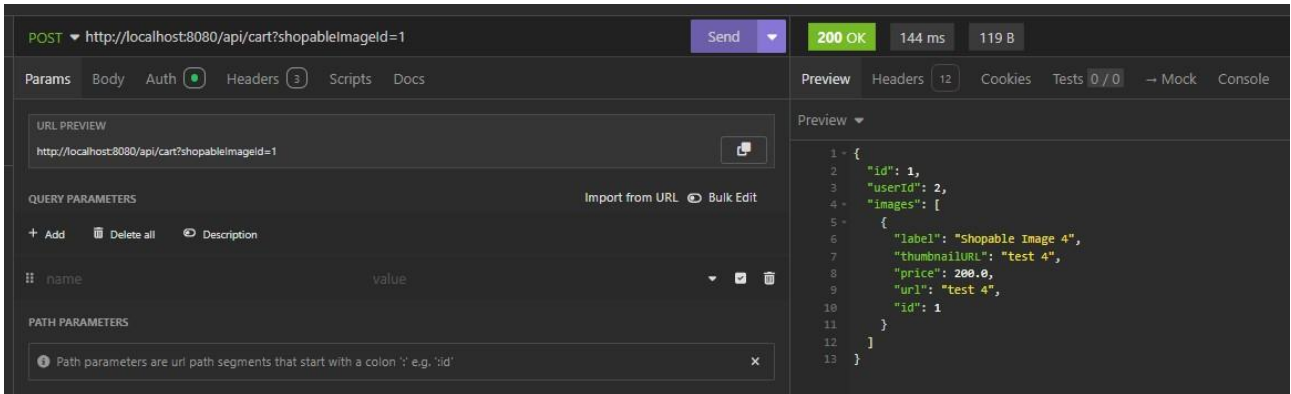
3. Login con utenza user, in modo da ottenere un token bearer nuovo



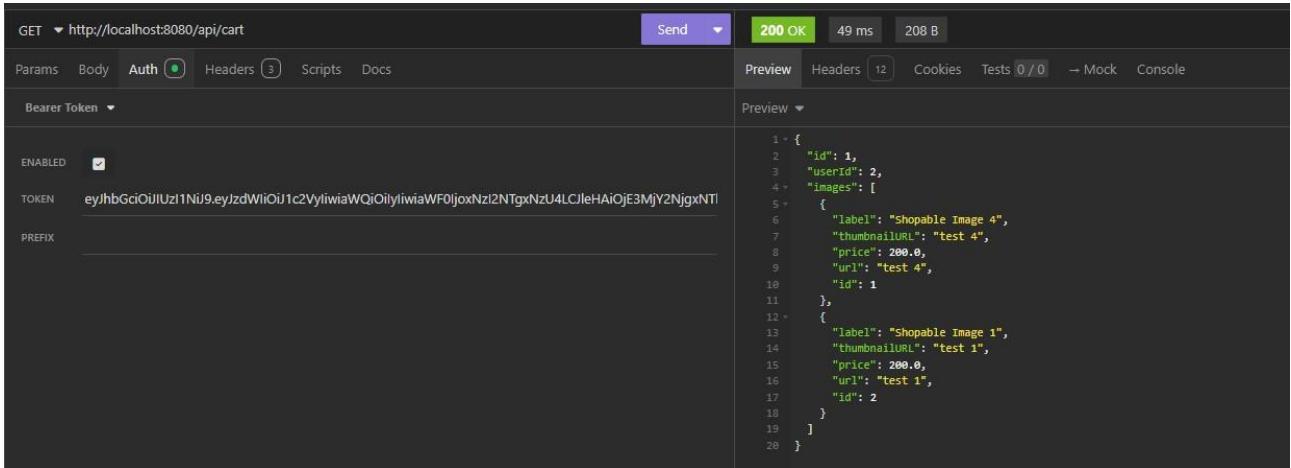
4. Simuliamo l'apertura dello shop, chiedendo al back-end l'elenco di tutte le immagini acquistabili



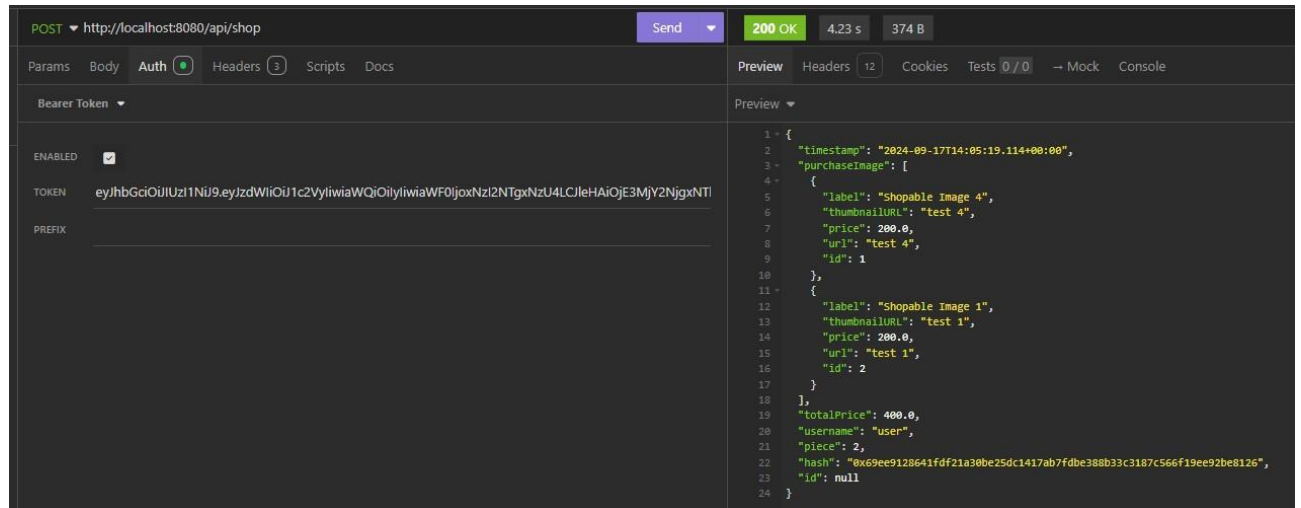
5. Aggiungiamo al carrello le immagini con ID 1 e 2



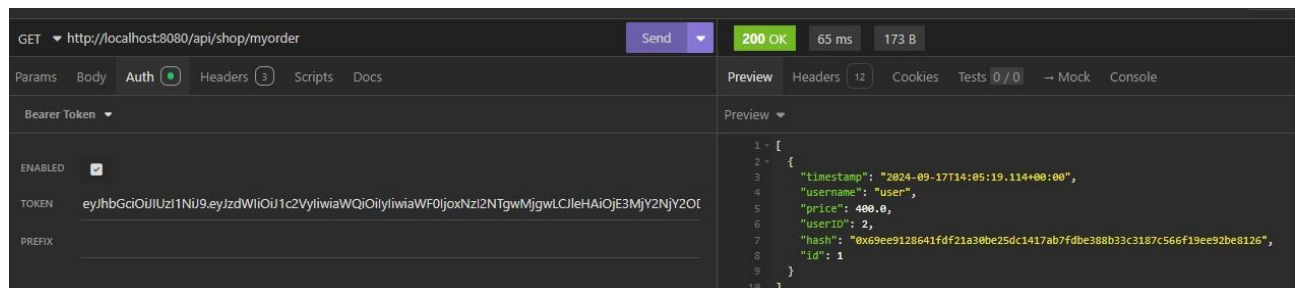
6. Verifichiamo che nel carrello siamo presenti le immagini con ID 1 e 2



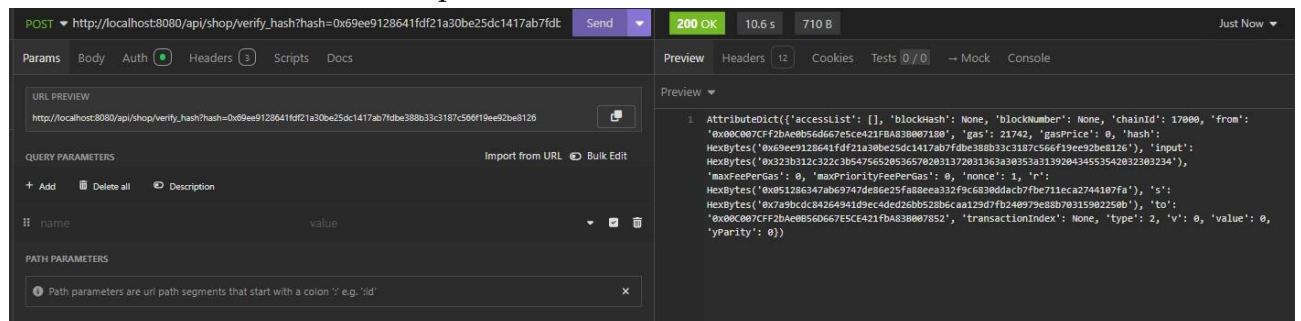
7. Effettuiamo l'ordine



8. Per riprova verifichiamo che l'ordine sia stato salvato sul DB. Possiamo notare che è stato salvato anche l'hash della nuova transazione



9. Tramite l'utenza amministratore possiamo effettuare una verifica sulla veridicità del blocco



10. Facendo il decode dell'array *input* possiamo vedere i dati salvati all'interno del blocco, che riportano *user id*, i due *image id*, e il *timestamp* della transazione

```
In [11]: runfile('C:/Users/Andrea/Documents/SWAM/workspace/myportfolio/hash_verify.py', wdir='C:/Users/Andrea/
Documents/SWAM/workspace/myportfolio')
b'2;1,2;;Tue Sep 17 16:05:19 CEST 2024'
```

API

In questa sezione si riporta l'elenco di tutte gli endpoint disponibili sul backend

AuthController					
path	method	parameter	auth	return	commento
/api/auth/signup	POST	@RequestBody LoginRequest		UserInfoResponse + Cookie Token JWT	Registrazione
/api/auth/signin	POST	@RequestBody SignupRequest		Message	Login
/api/auth/signout	POST	Cookie Token JWT		Message	Logout
/api/auth/checkmail	GET	@RequestParam("email")		Message	Verifica la presenza di una mail già associata ad un'utenza registrata
/api/auth/checkusername	GET	@RequestParam("username")		Message	Verifica la presenza di una username già associata ad un'utenza registrata
/api/auth/updatepassword	POST	Cookie Token JWT, @RequestBody UpdatePasswordRequest	User Role	Message + Cookie Token JWT	Cambio password

CartController					
path	method	parameter	auth	return	commento
/api/cart	POST	Cookie Token JWT, @RequestParam Long	User Role	CartDTO	Aggiunge una ShopableImage al carrello
/api/cart	GET	Cookie Token JWT	User Role	CartDTO	Restituisce il carrello
/api/cart	DELETE	Cookie Token JWT, @RequestParam Long shopableImageId	User Role	CartDTO	Elimina un ShopableImage dal carrello passando con parametro l'id
/api/cart/empty	DELETE	Cookie Token JWT	User Role	Message	Svuota il carrello

ImageController					
path	method	parameter	auth	return	commento
/api/image	GET	Cookie Token JWT, @RequestParam Long id	User Role, Admin Role	MediaType.IMAGE_PNG	generando il watermark a runtime, fornito in input l'id
/api/image	POST	Cookie Token JWT, @RequestBody List<ImageDTO>	Admin Role	Message	Aggiunta nuova immagine, è necessario passare un Id di un Work valido
/api/image	DELETE	Cookie Token JWT, @RequestParam Long id	Admin Role	Message	Cancella immagine dato un Id
/api/image	PUT	Cookie Token JWT, @RequestBody ImageDTO	Admin Role	Message	Aggiorna parametri immagine
/api/image/thumb	GET	Cookie Token JWT, @RequestParam Long id	User Role, Admin Role	MediaType.IMAGE_PNG	ritorna la thumbnail di una ImageProject fornito l'id

ShopController					
path	method	parameter	auth	return	commento
/api/shop/allorder	GET	Cookie Token JWT	Admin Role	List<SalesOrderDTO>	Restituisce l'elenco di tutti gli ordini fatti da tutti gli utenti
/api/shop/myorder	GET	Cookie Token JWT	User Role	List<SalesOrderDTO>	Restituisce l'elenco di tutti gli ordini fatti dall'utente che ne fa richiesta
/api/shop/	POST	Cookie Token JWT	User Role	DetailsSalesOrderDTO	Esegue l'acquisto in base alle immagini presenti nel carrello
/api/shop/orderdetail	GET	Cookie Token JWT, @RequestParam Long id	User Role	DetailsSalesOrderDTO	Dato l'id di un ordine restituisce i dettagli
/api/shop/verifyhash	POST	Cookie Token JWT, @RequestParam String hash	Admin Role	Message or String	Dato un hash ne verifica l'autenticità

ShopableImageController					
path	method	parameter	auth	return	commento
/api/shopableimage/all	GET		User Role	List<SimpleShopableImageDTO>	Restituisce tutte le thumbnail shopableimage che possono essere acquistate
/api/shopableimage	GET	Cookie Token JWT, @RequestParam Long id	User Role	MediaType.IMAGE_PNG	Restituisce la ShopableImage, generando il watermark a runtime, fornito in input l'id
/api/shopableimage/issold	GET	Cookie Token JWT, @RequestParam Long id	User Role	Message	verificare se una certa shopableimage è acquistabile
/api/shopableimage	POST	Cookie Token JWT, @RequestBody ShopableImageDTO	Admin Role	ShopableImageDTO	Aggiunta ShopableImage
/api/shopableimage	PUT	Cookie Token JWT, @RequestParam Long id	Admin Role	ShopableImageDTO	Aggiornamento parametri ShopableImage
/api/shopableimage	DELETE	Cookie Token JWT, @RequestParam Long id	Admin Role	Message	Cancellazione ShopableImage
/api/shopableimage/thumb	GET	Cookie Token JWT, @RequestParam Long id	User Role, Admin Role	MediaType.IMAGE_PNG	ritorna la thumbnail di una ShopableImage fornito l'id

ThumbnailController					
path	method	parameter	auth	return	commento
/api/thumbnail	POST	Cookie Token JWT, @RequestParam Long id	Admin Role	Message	Dato l'id di un Work genere le thumbnail delle immagini presenti al suo interno

UserController					
path	method	parameter	auth	return	commento
/api/user	GET	Cookie Token JWT	User Role	UserPersonalDetailsDTO	Restituisce anagrafica utente
/api/user	PUT	Cookie Token JWT, @RequestBody UserPersonalDetailsDTO	User Role	UserPersonalDetailsDTO	Aggiornamento anagrafica utente
/api/user	DELETE	Cookie Token JWT, @RequestParam String username	Admin Role	Message	permette la cancellazione di un utente dato un username (disabilita l'utenza per non perdere i suoi dati)

WorkController					
path	method	parameter	auth	return	commento
/api/work/all	GET	Cookie Token JWT	User Role, Admin Role	Set<WorkDTO>	restituisce tutti i work per cui un utente ha l'accesso in visualizzazione
/api/work	POST	Cookie Token JWT, @RequestBody WorkDTO	Admin Role	WorkDTO	Aggiunta nuovo Work
/api/work	DELETE	Cookie Token JWT, @RequestParam Long id	Admin Role	Message	Cancellazione Work
/api/work	PUT	Cookie Token JWT, @RequestBody WorkDTO	Admin Role	WorkDTO	Modifica Work
/api/work	GET	Cookie Token JWT, @RequestParam Long id	User Role, Admin Role	DetailsWorkDTO	Restituisce le image contenute in un work passando l'id di un
/api/work/visble-work	PATCH	@RequestParam Long userId, @RequestBody ArrayList<Long>	Admin Role	Message	di Work, aggiorna i diritti di visualizzazione