Immagine che contiene schizzo, disegno, arte, bianco e nero

Descrizione generata automaticamente

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

Sommario

[Introduzione 3](#_heading=h.gjdgxs)

[Obiettivi e intenti dell'elaborato 3](#_heading=h.30j0zll)

[Analisi dei requisiti 4](#_heading=h.1fob9te)

[Vincoli 4](#_heading=h.3znysh7)

[Requisiti non funzionali 4](#_heading=h.2et92p0)

[Requisiti funzionali 4](#_heading=h.tyjcwt)

[Analisi di progettazione 6](#_heading=h.3dy6vkm)

[Gestione e organizzazione delle immagini 6](#_heading=h.1t3h5sf)

[Accesso e permessi utente 6](#_heading=h.4d34og8)

[Visualizzazione e Copyright delle Immagini 6](#_heading=h.2s8eyo1)

[Sezione Shop (Estensione del Progetto) 7](#_heading=h.17dp8vu)

[Considerazioni sulla Scalabilità e Manutenibilità 7](#_heading=h.3rdcrjn)

[Analisi diagrammi di progetto 7](#_heading=h.26in1rg)

[**UML Diagram** 8](#_heading=h.lnxbz9)

[**Deployment Diagram** 10](#_heading=h.35nkun2)

[**Sequence Diagram** 10](#_heading=h.1ksv4uv)

[Descrizione scelte implementative 14](#_heading=h.44sinio)

[Descrizione dei package 14](#_heading=h.z337ya)

[Descrizione componenti principali 15](#_heading=h.3j2qqm3)

[Domain Model 16](#_heading=h.2jxsxqh)

[Implementazione lato JAVA 16](#_heading=h.1y810tw)

[Implementazione lato DB 17](#_heading=h.4i7ojhp)

[Implementazione front-end 17](#_heading=h.2xcytpi)

[Sezione shop e blockchain 17](#_heading=h.1ci93xb)

[API 17](#_heading=h.3whwml4)

[Analisi api con Flusso - tabella di copertura 17](#_heading=h.2bn6wsx)

# Introduzione

L'idea del progetto nasce da una richiesta di un amico fotografo che aveva il problema di condividere 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 accumunati 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

* 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.
* 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

* Architettura del Sistema
  + L'architettura del sistema deve essere basata sul modello MVC.
  + Il back-end deve essere sviluppato utilizzando JakartaEE con il framework Spring.
  + Il front-end deve essere sviluppato utilizzando Vue JS e Bootstrap.
  + L'architettura deve garantire la portabilità del sistema.
* Gestione del Copyright
  + Il sistema deve garantire il rispetto del copyright sulle opere

## Requisiti funzionali

* Tipologie utenze e azioni permesse
  + L'applicazione deve prevedere 2 tipolgie di utenza: Admin e User
  + L'utenza con Role User deve avere la possibilità di visualizzare i work a cui gli è stato fornito l'accesso
  + L'utenza con Role Admin deve aver la possibilità di visualizzare tutti i work disponibili
  + L'utenza con Role Admin deve aver la possibilità di accedere ai pannelli di amministrazione sia della sezione Galleria che Shop
  + L'utenza con Role User deve accedere e fare acquisti nella sezione Shop
* Gestione pannello di controllo dell'amministratore
  + L'amministratore deve poter accedere a un pannello di controllo.
  + Dal pannello di controllo, l'amministratore deve poter caricare risorse (immagini).
  + Gestione Permessi di Lettura
  + Dal pannello di controllo, l'amministratore deve poter gestire i permessi di lettura per i singoli "work".
* Gestione immagini
  + 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).
  + L'applicativo deve supportare il caricamento, la catalogazione e il recupero di risorse immagine nei formati PNG, JPEG, JPG e BMP.
  + Il sistema deve generare automaticamente le miniature (thumbnail) per ogni immagine caricata per ottimizzare i tempi di caricamento delle pagine.
  + 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.
  + 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
  + Il sistema deve permettere di applicare in modo statico un watermark sulle thumbnail
* Gestione fine dei permessi di autorizzazione e accesso
  + Gli utenti devono registrarsi alla piattaforma per accedere alle risorse.
  + Ogni utente deve avere autorizzazioni mirate su cartelle specifiche.
  + Alcune immagini utilizzate nella homepage devono avere visibilità pubblica senza necessità di login.
  + L'accesso alle foto private deve avvenire solo dopo login dell'utente.
  + L'utente deve poter visualizzare solo le foto per cui ha i diritti di accesso.
* Componenti Java riusabili
  + Il sistema deve includere componenti Java riusabili per la generazione automatica delle miniature delle immagini.
  + Il sistema deve consentire l'aggiunta di un watermark personalizzato alle risorse immagine su richiesta (on demand).
* Sezione Shop e gestione acquisti
  + L'applicativo deve prevedere una sezione "Shop" dove saranno presenti alcune immagini selezionate dall'amministratore, simili a NFT.
  + Gli utenti devono poter acquistare le immagini presenti nella sezione "Shop".
  + Gli acquisti devono essere registrati su un sistema blockchain utilizzando le API fornite da un provider di tecnologia blockchain.

# 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). Invece, 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 recupere solo stringhe testuali, sia 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 Facilità 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 mantenibile. 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 di definiscono modelli concettuali (UML, Use Case, Sequence …) 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.
* ImageProject: rappresenta un'immagine associata a un progetto.
* ShopableImage: rappresenta un'immagine disponibile per l'acquisto all’interno dello shop.
* Cart: rappresenta il carello 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

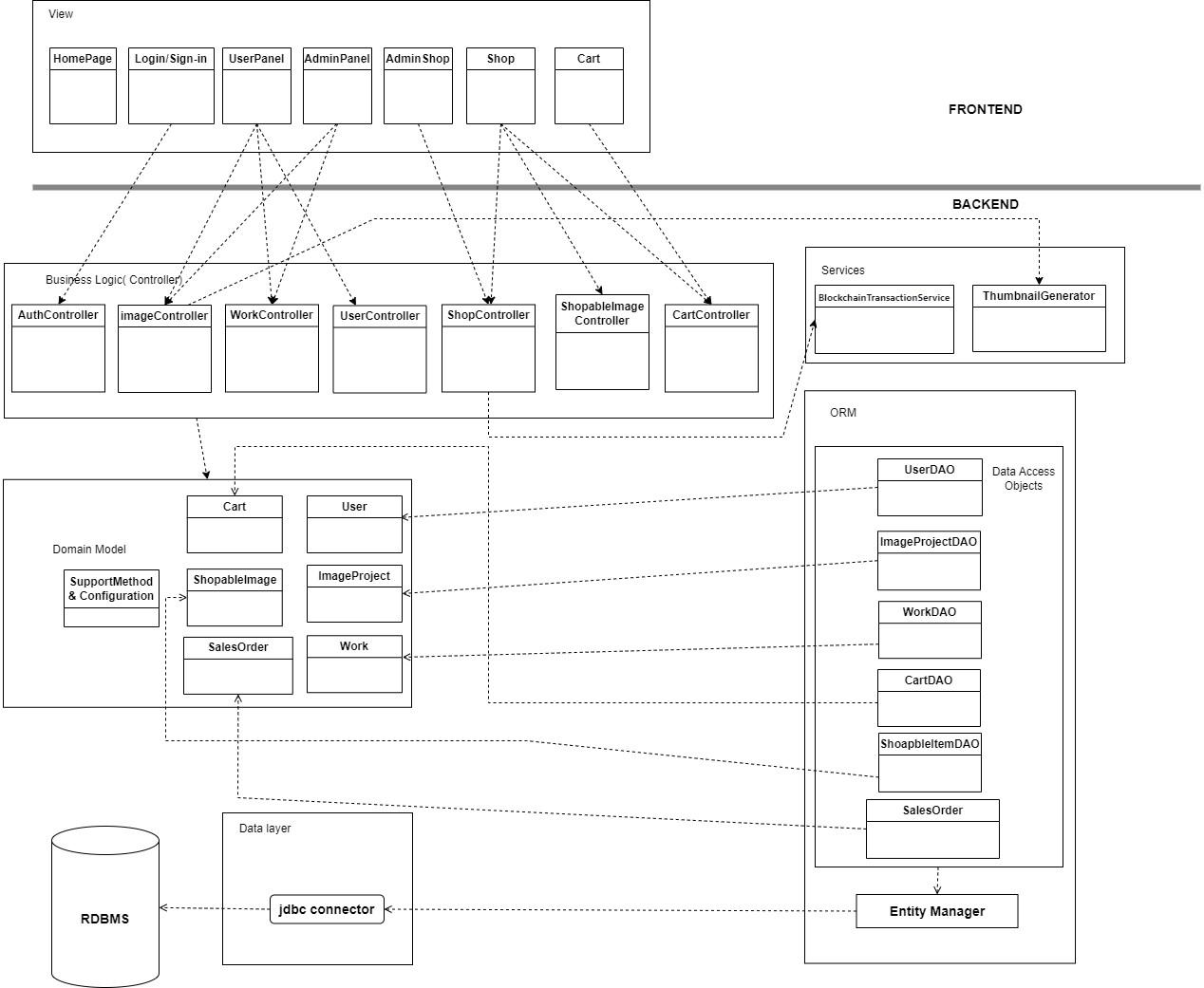
Immagine che contiene schermata, diagramma, design

Descrizione generata automaticamente

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
* **ShopableImage:** 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.  
  le informazioni che memorizza questa classe dovranno essere 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.

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).

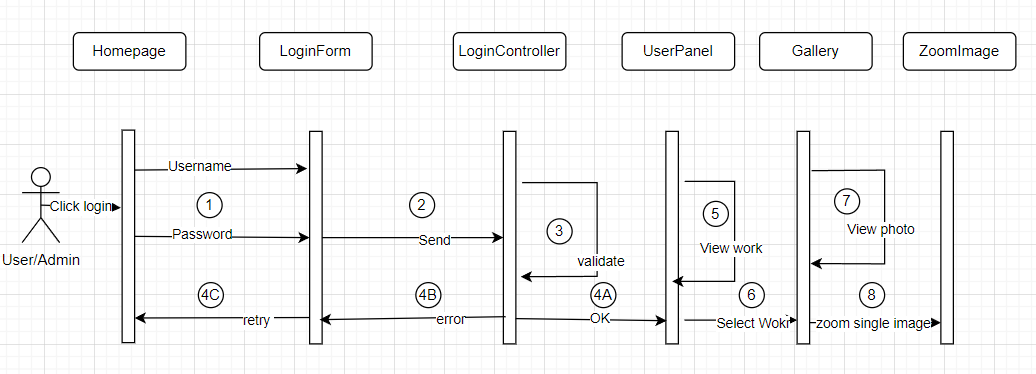
Grazie a questa suddivisione, si ottengono due vantaggi molto importanti:

* **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.

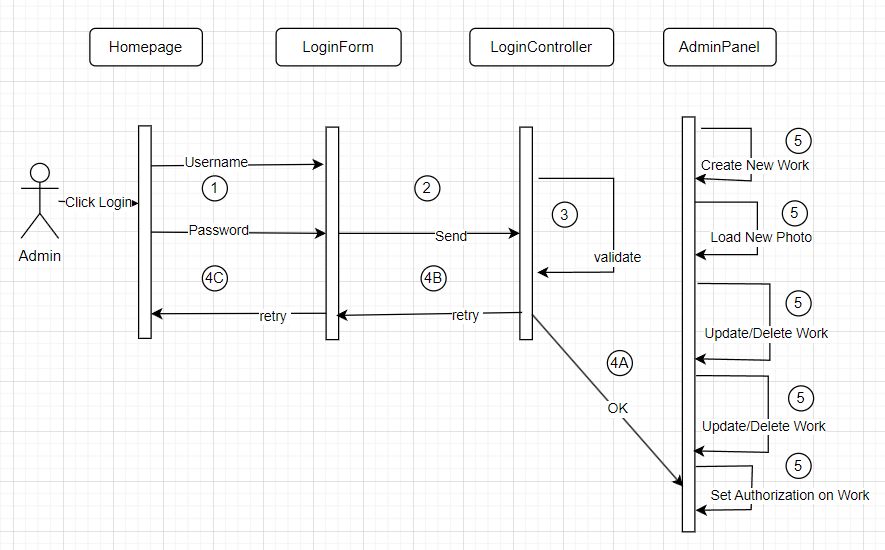
## Sequence Diagram

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

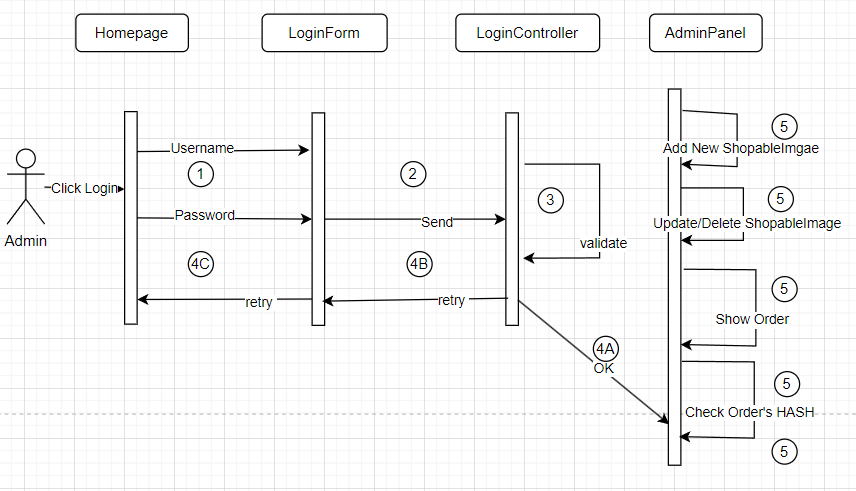
Da qui potrà accedere in visualizzazione ai Work e alle relative immagini contenute in essi, per i quali avrà ottenuto l’autorizzazione della visualizzazione.



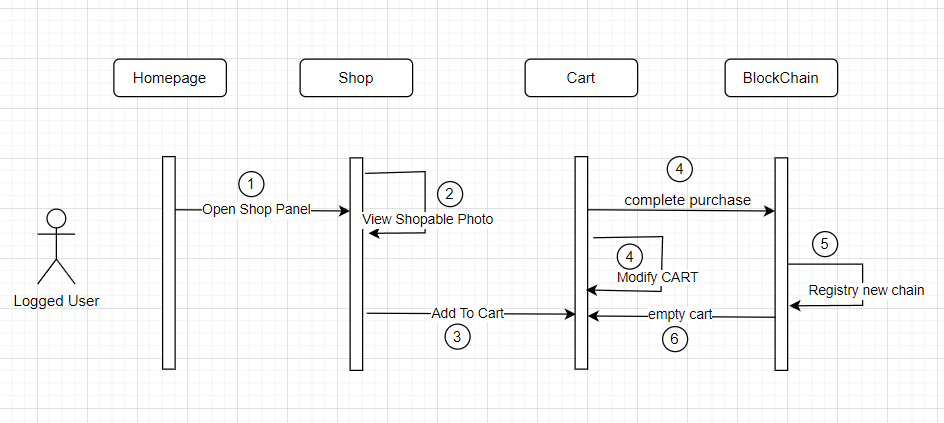
L’admin dal proprio pannello di gestione potrà creare di nuovi Work, avrà la completa gestione di aggiornamento e cancellazione dei Work, potrà caricare immagini specificando a quale Work appartengono. Avrà la completa gestione di aggiornamento e cancellazione delle singole immagini e potrà settare le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



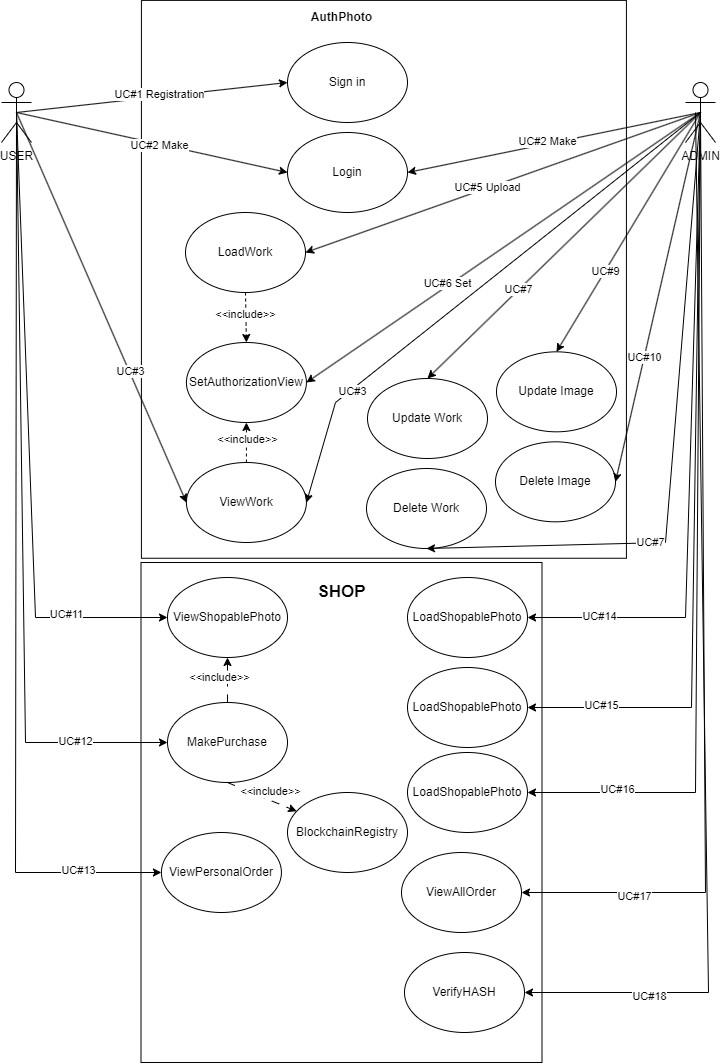
L’admin dalla View AdminShop potrà caricare nuove ShopableImage, cancellare/aggiornare quelle già in vendita, potrà visualizzare tutti gli ordini fatti e verificare la veridicità dell’Hash delle transizioni di vendita.



L'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 LoadWork: L'utente carica un lavoro fotografico.
* UC#4 SetAuthorizationView: L'utente richiede l'autorizzazione per visualizzare un lavoro.
* UC#5 Upload: L'amministratore carica una nuova immagine.
* UC#6 Update Work: L'amministratore aggiorna un lavoro esistente.
* UC#7 Delete Work: L'amministratore elimina un lavoro.
* UC#8 Update Image: L'amministratore aggiorna un'immagine esistente.
* UC#9 Delete Image: L'amministratore elimina un'immagine.
* UC#10 Set: L'amministratore concede l'autorizzazione per visualizzare un lavoro.
* UC#11 ViewWork: L'utente visualizza un lavoro
* UC#12 ViewShopablePhoto: L'utente visualizza le foto disponibili per l'acquisto.
* UC#13 MakePurchase: L'utente effettua un acquisto.
* UC#14 ViewPersonalOrder: L'utente visualizza i propri ordini.
* UC#15 LoadShopablePhoto: L'amministratore carica le foto disponibili per la vendita.
* UC#16 LoadShopablePhoto: L'amministratore carica le foto 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.

Nella sezione *“Analisi API - tabella di copertura”* verranno analizzati ognuno di questi Use Case, andando vedere quale API o la sequenza di API da chiamare per verificarlo.

# 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.

Immagine che contiene testo, schermata, Carattere, design

Descrizione generata automaticamente

* **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 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, gestiscono 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, come configurare il sistema di sicurezza per l'applicazione web. Questa classe estende o implementa 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 watemark. 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.

Del “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

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 presenta poche righe di codice, è una delle classi principali del progetto. Si occupa di fare il discovery dei vari componenti del boostrap 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. 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 di convenienza che esegue automaticamente il bootstrap di un'applicazione Spring da un metodo main statico.

1. **Configura l'applicazione**:

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

1. **Scansione dei componenti**: scansiona i pacchetti per trovare i componenti, le configurazioni e i servizi annotati con @Component, @EnableJpaRepositories, @EntityScan
2. **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

1. **Avvia l'ApplicationContext**: è il container principale di Spring che gestisce i bean e le loro dipendenze.
2. **Avvia il server web integrato**: avvia automaticamente il server web Tomcat incorporato per servire l'applicazione.



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.

## it.myportfolio.model

### Work



### ImageProject



### ShopableImage



### User



### Role

### ERole

### Cart

### SalesOrder

## 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:

* Tramire **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.



Snippet 2: metodo per fare il mapping 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).



Snippet 3: metodo per fare il mapping Entity -> DTO

## Descrizione componenti di utility

Oltre ai componenti visti in precedenza che rappresentano i pilastri del progetto sono stare realizzate anche le due classi di “servizio” BlockchainTransactionService e ThumbnailGenerator, che fornisco 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. Estrae 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



Snippet 4: funziona makeThumbnail che si occupa di generare e salvare nel path indicato, una nuova immagine ridotta del 30% rispetto all’orignale. La partedi verifica della presenza dell’immagine, la verifica e/o creazione del path di destinazione dell’immagine creata è omesso.



Snippet 5: 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”*

# **Descrizione** scelte implementative

Nella sezione "Descrizione scelte implementative" di una relazione tecnica, dovresti fornire dettagli chiari e motivati sulle decisioni chiave prese durante il processo di sviluppo. Questa sezione dovrebbe spiegare le ragioni delle scelte tecniche, evidenziando come esse contribuiscono al raggiungimento degli obiettivi del progetto. Ecco alcuni punti che potresti includere:

### Descrizione delle Scelte Implementative

1. **Architettura del Sistema:**
   * **Scelta dell'Architettura:** Descrivere perché è stata scelta una specifica architettura (ad esempio, architettura MVC, architettura a microservizi).
   * **Motivazioni:** Spiegare i benefici di questa architettura in termini di scalabilità, manutenibilità e performance.
2. **Tecnologie e Strumenti:**
   * **Backend:** Motivare la scelta di JakartaEE e Spring Framework per il backend, sottolineando vantaggi come l'integrazione con i servizi esistenti, la robustezza e la facilità di gestione delle dipendenze.
   * **Frontend:** Spiegare perché sono stati scelti Vue.js e Bootstrap per il frontend, evidenziando aspetti come la responsività, la facilità di utilizzo e la comunità di supporto.
   * **Database:** Indicare il database scelto (ad esempio, PostgreSQL, MySQL) e motivare la scelta in base a performance, scalabilità e funzionalità supportate.
3. **Gestione delle Risorse:**
   * **Memorizzazione delle Immagini:** Spiegare la decisione di non caricare le immagini nel database come BLOB, ma di utilizzare URL salvati nel database per gestire i riferimenti ai file.
   * **ACL Personalizzate:** Descrivere la scelta di creare una lista di controllo degli accessi (ACL) ad hoc per gestire i permessi di visualizzazione delle immagini.
4. **Sicurezza:**
   * **Autenticazione e Autorizzazione:** Descrivere le tecniche utilizzate per garantire la sicurezza degli accessi (ad esempio, OAuth2, JWT).
   * **Gestione dei Permessi:** Spiegare come vengono gestiti i permessi a livello di utente e di risorsa per garantire che solo gli utenti autorizzati possano accedere a determinate immagini o funzionalità.
5. **Performance e Ottimizzazione:**
   * **Generazione delle Thumbnail:** Spiegare come e perché è stata implementata la generazione di thumbnail lato server per migliorare i tempi di caricamento delle pagine.
   * **Caching:** Descrivere eventuali strategie di caching implementate per migliorare le performance del sistema.
6. **Estendibilità e Modularità:**
   * **Componenti Riutilizzabili:** Indicare quali componenti Java sono stati progettati per essere riutilizzabili e modulabili, e come ciò facilita l'aggiunta di nuove funzionalità in futuro.
   * **API:** Spiegare l'uso di API per facilitare l'integrazione con altre applicazioni o per permettere la creazione di app native per Android e iOS.
7. **Gestione dei Dati:**
   * **Persistenza:** Descrivere il metodo di persistenza dei dati, incluse le scelte relative all'ORM (ad esempio, Hibernate) e alla configurazione del JDBC.
   * **Transazioni:** Spiegare come vengono gestite le transazioni per garantire la coerenza dei dati.
8. **Funzionalità Avanzate:**
   * **Watermarking:** Descrivere l'implementazione del watermarking delle immagini lato client per proteggere i diritti d'autore.
   * **Blockchain:** Spiegare l'integrazione con un sistema blockchain per la gestione delle transazioni legate alla vendita delle immagini come opere uniche.

### Esempio di "Descrizione delle Scelte Implementative"

#### Architettura del Sistema

Abbiamo scelto un'architettura basata su MVC per garantire una separazione chiara delle responsabilità tra il modello, la vista e il controller. Questa scelta facilita la manutenibilità e l'estensibilità del sistema. L'architettura a microservizi è stata considerata, ma scartata per questo progetto specifico a causa della sua complessità aggiuntiva, che non era giustificata dalla scala del progetto.

#### Tecnologie e Strumenti

Per il backend, abbiamo selezionato JakartaEE con il framework Spring per sfruttare la sua robustezza e la vasta gamma di funzionalità integrate. Per il frontend, Vue.js è stato scelto per la sua facilità d'uso e l'efficacia nella costruzione di interfacce utente reattive. Bootstrap è stato integrato per garantire che l'applicazione fosse completamente responsive.

#### Gestione delle Risorse

Abbiamo deciso di non caricare le immagini nel database come BLOB per evitare problemi di performance. Invece, le immagini sono memorizzate su un NAS e i loro URL sono registrati nel database. Questo approccio riduce il carico sul database e facilita la gestione dei file multimediali.

#### Sicurezza

L'autenticazione e l'autorizzazione sono gestite tramite OAuth2, garantendo un alto livello di sicurezza per l'accesso alle risorse. Le ACL personalizzate permettono una gestione granulare dei permessi, assicurando che solo gli utenti autorizzati possano accedere alle immagini specifiche.

#### Performance e Ottimizzazione

La generazione delle thumbnail avviene lato server per migliorare i tempi di caricamento delle pagine. Questa scelta riduce il tempo necessario per visualizzare le immagini nelle gallerie e nelle anteprime, migliorando l'esperienza utente.

#### Estendibilità e Modularità

Abbiamo progettato componenti Java riutilizzabili per operazioni comuni come la gestione delle immagini e dei permessi. Questi componenti possono essere facilmente estesi per aggiungere nuove funzionalità senza compromettere quelle esistenti. Inoltre, l'uso di API facilita la creazione di app native per Android e iOS, migliorando la portabilità del sistema.

#### Gestione dei Dati

La persistenza dei dati è gestita tramite Hibernate, che facilita la mappatura degli oggetti Java alle tabelle del database. Questo approccio semplifica la gestione delle relazioni tra le entità e garantisce la coerenza dei dati. Le transazioni sono gestite per garantire che tutte le operazioni di lettura e scrittura siano atomiche e consistenti.

#### Funzionalità Avanzate

Il watermarking delle immagini viene eseguito lato client per proteggere i diritti d'autore senza dover memorizzare versioni duplicate delle immagini. Inoltre, un sistema blockchain è integrato per registrare le transazioni legate alla vendita delle immagini, garantendo trasparenza e sicurezza nelle operazioni di acquisto.

### Conclusione

La sezione "Descrizione scelte implementative" deve fornire una visione chiara e dettagliata delle decisioni tecniche prese durante il progetto. Deve mostrare come queste scelte supportano gli obiettivi del progetto e rispondono alle esigenze degli stakeholder, assicurando al contempo che il sistema sia robusto, sicuro, scalabile e manutenibile.

# Implementazione lato DB

I tool utilizzati per lo strato di persistenza sono:

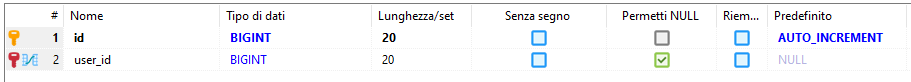
* ORM: Hibernate 6.4.4 Final
* RDBMS: MariaDB 11.2.
* Tool di amministrazione: HeidiSQL 12.3.0.6589

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](https://it.wikipedia.org/wiki/Persistenza_(informatica)) dei dati sul database dei modelli definiti come oggetti Java.

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

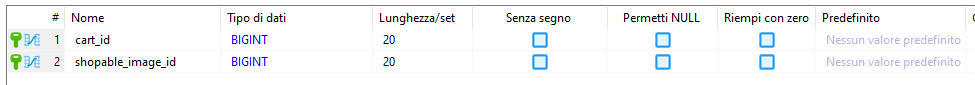


Segue adesso un dettaglio delle singole tabelle, riportate nello stesso ordine dell’immagine precedente, con alcuni commenti nelle tabelle più interessanti

**cart**

**cart\_shopable\_image**

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



**hibernate\_sequences**

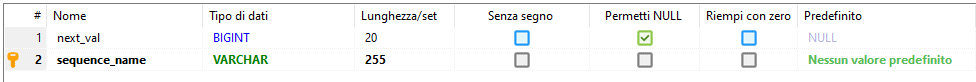
La tabella hibernate\_sequences viene creata automaticamente da Hibernate quando si utilizza la strategia di generazione delle chiavi primarie GenerationType.TABLE, che permette di generare identificatori unici per le entità gestite da Hibernate.  
Quando si decide di utilizzare GenerationType.TABLE viene utilizzata una tabella specifica per tenere traccia dei numeri di sequenza utilizzati per generare gli ID delle entità. Questo approccio è utile quando si desidera avere un controllo personalizzato su come vengono generati gli ID o quando il database non supporta altre strategie di generazione come AUTO o SEQUENCE.

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.

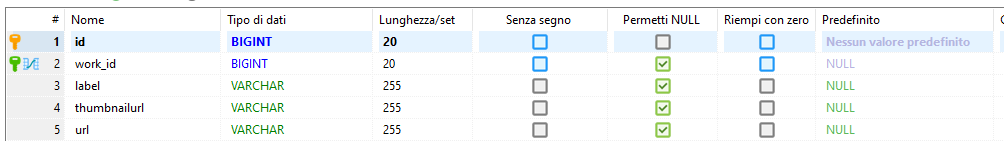
Hibernate: create table hibernate\_sequences (next\_val bigint, sequence\_name varchar(255) not null, primary key (sequence\_name)) engine=InnoDB

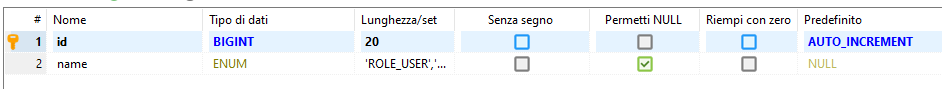


**image**

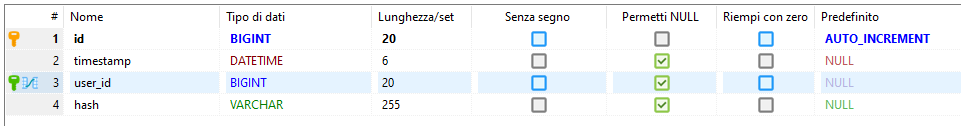
Su questa tabella vengono salvate le entità create tramite il model ImageProject, il parametro work\_id specifica a quale Work deve essere associata l’immagine.

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

**roles**



**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”* 

**shopable\_image**

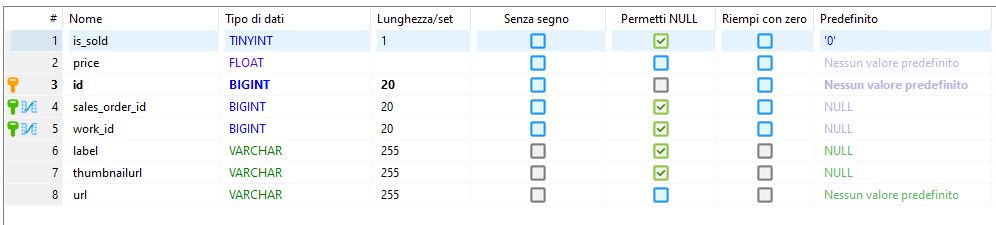
Questa tabella memorizza le immagini che l’admin ha deciso di mettere in vendita all’interno dello shop. Poiché la classe (model) ShopableImage è realizzata sfruttando l’ereditarietà a partire dal model ImageProject e poiché è stata utilizzata l'annotazione @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS),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

Non esiste una tabella condivisa che rappresenti la superclasse astratta (se presente), e ogni tabella contiene tutte le colonne necessarie per rappresentare completamente le istanze delle rispettive sottoclassi.

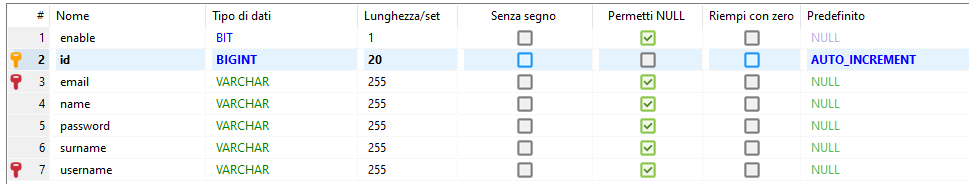
Oltre ai campi ereditati, questa classe ha in più i seguenti parametri:

* ***price:*** che indica il prezzo di vendita
* ***sales\_order\_id:*** se compilato indica l’id dell’ordine con il quale è stata acquista l’immagine
* ***is\_sold:*** indica se quell’immagine è stata acquistata o meno

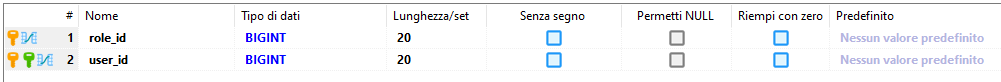
Quando viene aggiunta una nuova immagine alla vendita, questa farà parte del Work con ID 0.



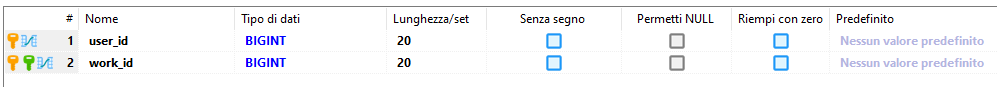
**user**



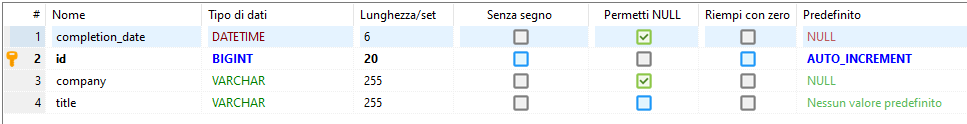
**user\_roles**



**visible\_work**



**work**

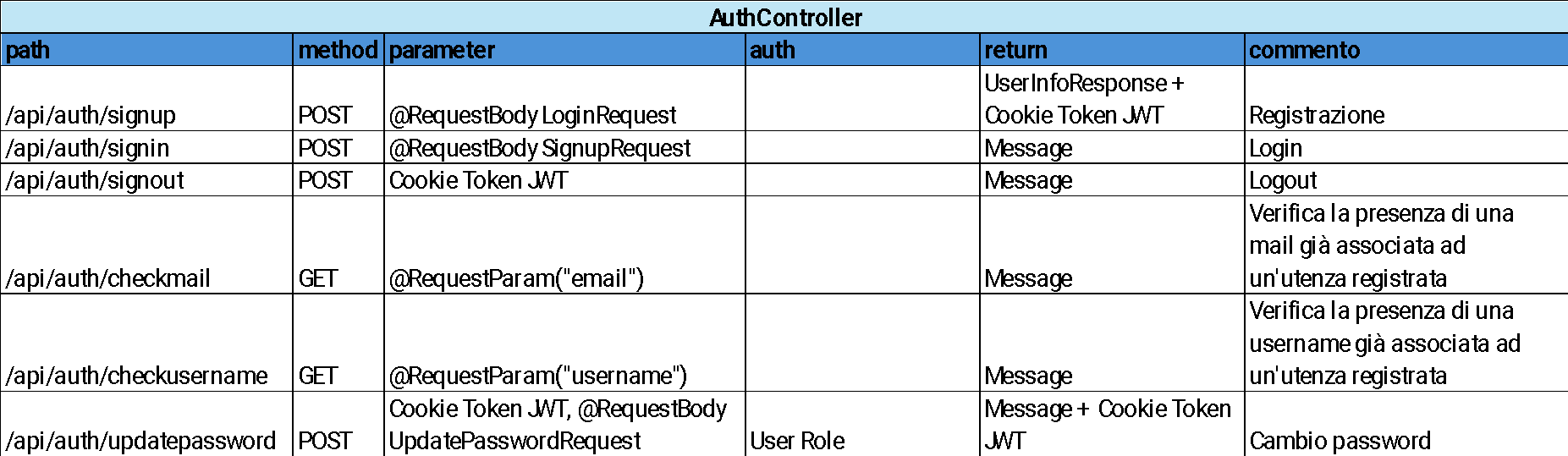


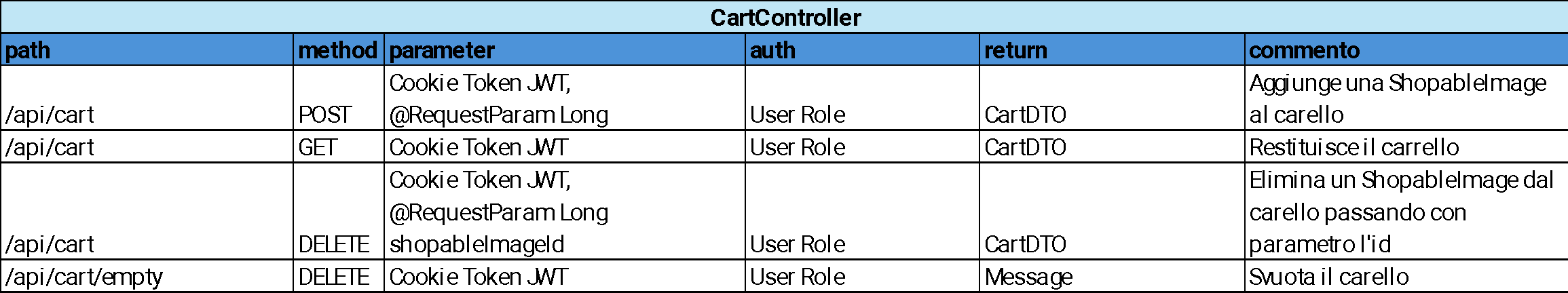
# Implementazione front-end

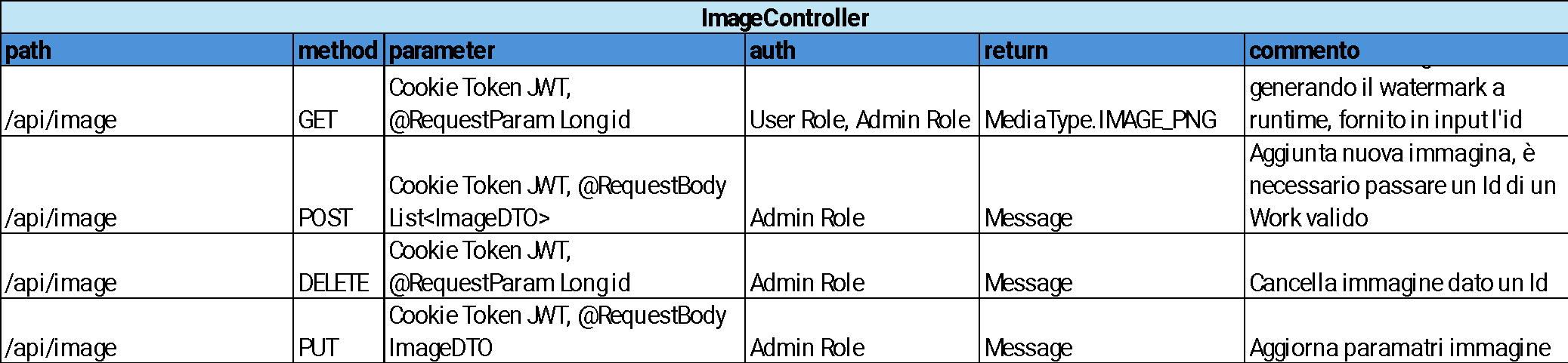
# Sezione shop e blockchain (x)

-circa la parte dedicata alla Blockchain, ricordati poi di documentare bene la piattaforma che selezionerai per la certificazione del dato e le API che adotterai).

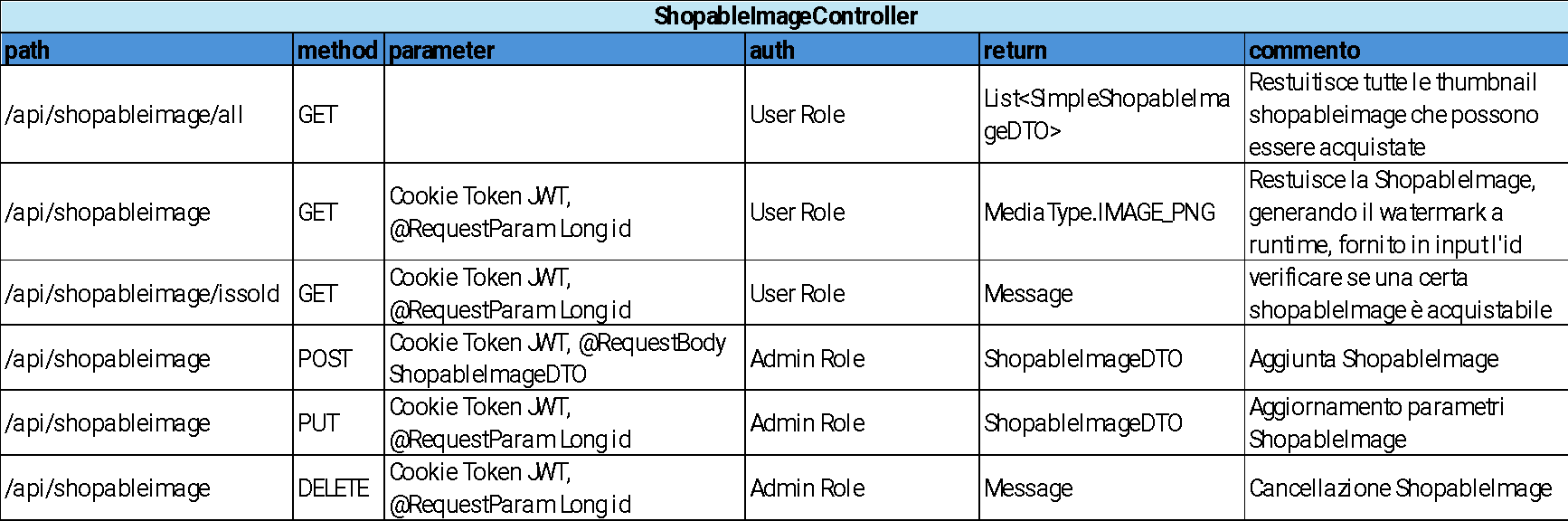
# API

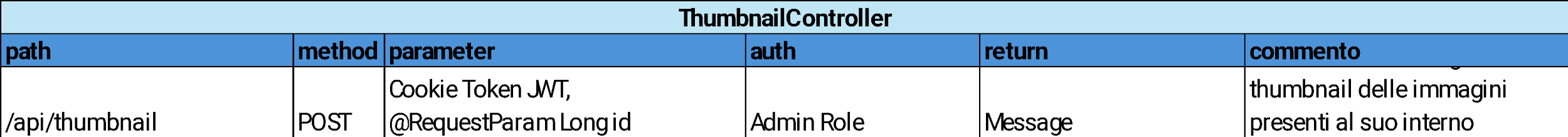


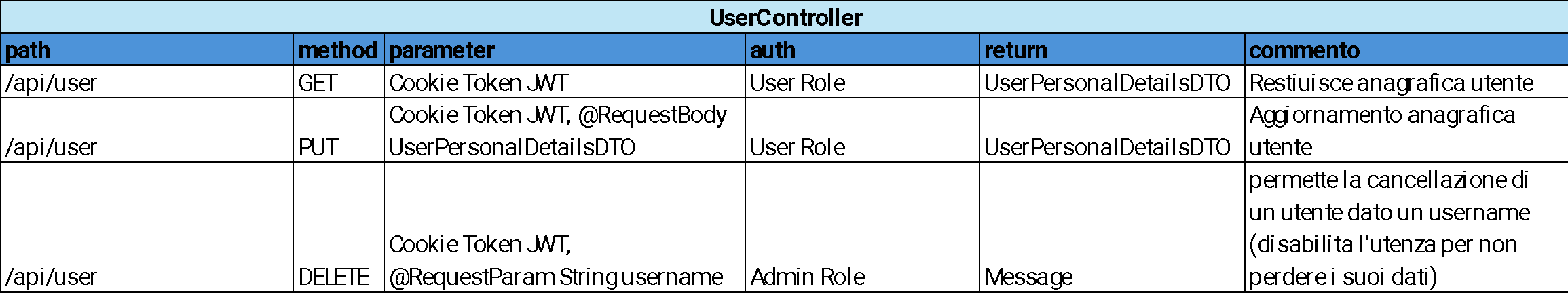


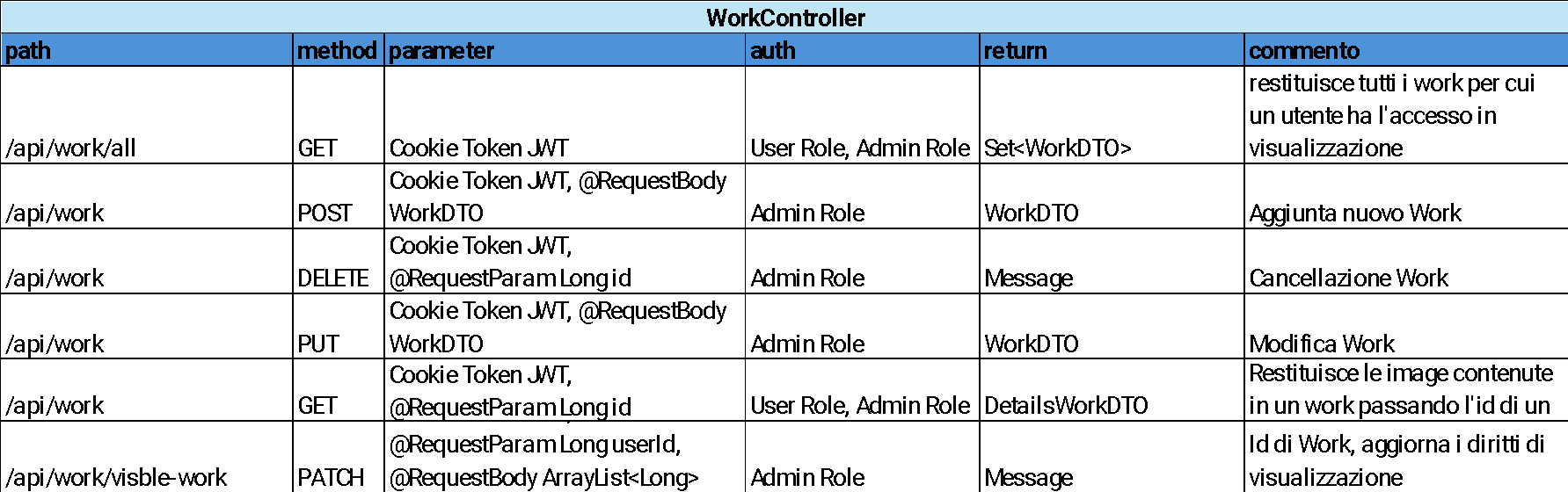












# Analisi API - tabella di copertura

- nei diagrammi dei casi d'uso ti consigliamo di indicare per ciascuno di essi un identificativo (tipo UC#1) e anche di predisporre una "tabella di copertura" tra i casi d'uso e i requisiti stilati (i quali andranno anche essi numerati)